BETA
SIMULATION SOLUTIONS

physics on screen

ANSA
PRE PROCESSOR

Modular Model and Run Management

Table of Contents

1. Introduction

1.1. Beyond Includes Management

1.1.1. Problem description

In today's CAE simulations, complex structures that consist of several sub-assemblies are handled with the aid of "include files". This technology, supported by all major FEA solvers, enables the subdivision of a complete CAE model in smaller units, each being an individual solver keyword file, and the reassembly of the complete model by the solver during the simulation. The big advantage of this working method is found in the CAE model improvement process (looping phase), where individual analysts can focus on different areas of interest and only update the keyword files concerned, while the integrity of the complete CAE model is maintained.

Teams working with include files usually store the individual includes in a network location accessible by all team members. Certain rules must be followed for the folder structure and for the naming of the files, so that different people can store files in a way that other team members can spot them. In order for an analyst to make an iteration, one or more includes from this shared location would have to be copied either locally or in the same network location, in order to be modified with an editor. Depending on the type of change, the editor would either be the pre-processor or, for local changes in parameter values, even the text editor. After the creation of the new version of the includes, the main file that corresponds to the previous run iteration would be copied too, in order to be modified, usually with a text editor, so that its INCLUDE keywords point to the new versions of the includes. These new main files would be submitted to the solver and then the user would run some post-processing session in order to review the results interactively and create reports.

This whole procedure works very efficiently for experienced users who, usually assisted by custom scripts, generate new run iterations, submit them to the solver and get processed key-results in minimum user time. However, this process has several weaknesses and hides many pitfalls with the most important listed below:

- Keeping track of model and loadcase variants always requires the existence of some additional tool, capable of combining the proper variants of includes. This is often done in complex Excel files, whose processing and interpretation into solver main files always requires some scripts.
- Traceability in the evolution of the simulation model is difficult to achieve and relies heavily on the thoroughness of each individual. Keeping track of model changes and their effect on the results, along with information on when a simulation was conducted and by whom, usually requires complex Excel files, whose update always requires some scripts.
- It is quite often that the modifications made by an individual on include level, put the integrity of the complete model assembly at stake. However, validating the integrity of the complete model at each iteration is not a viable option, since it would require loading the complete model in the pre-processor.
- Any custom scripts that have been employed in order to verify the model assembly or create connecting includes on the fly, always require a big number of rules and conditions to be met at include level. Any deviation from the rules leads to errors in the complete model. However, verification of the integrity of each include requires the use of scripts.
- The comparison of results between different simulation versions is usually inefficient. The evolution of key-values is kept in custom log books but in order to get more information on the results, the user will have to spot the right reports within the folder structure. And then, in order to get information on the model differences between the different simulation versions, they will have to rely on the comments inserted by each individual.

- Finally, this way of data organization limits the possibilities of synergies between teams that study different attributes (e.g. crash, NVH, durability, CFD). It is very difficult, if not impossible, for an analyst in the NVH team to find a particular model created by the crash team.

All these shortcomings boil down to a couple of key deficiencies of the current practices.

1. Heavy reliance on the individual: Users need to make sure that they follow rules consistently. If these rules are not followed, traceability, model integrity and results assessment, are all put at stake. And learning to follow these rules requires extensive training.
2. Heavy reliance on scripts: The lack of an actual “system” to deal with the multiple challenges of simulations management leads to the inevitable development of custom scripts to carry out important steps of the process. However, scripts also need maintenance, and this always tends to be problematic.

1.1.2. BETA’s Simulation Run Management solution

The BETA Suite offers an integrated solution for comprehensive Simulation Run Management that leverages the flexibility and efficiency of modular model organization with the robustness offered by the simulation run composition tools that guarantee the integrity of the run after every modification on module level.

The unique characteristics of the BETA Simulation Run Management Environment are:

- It offers a complete solution for the management of the run complexity, including variants management on subsystem, simulation model and loadcase level.
- It enables traceability through the simulation data, allowing the analyst to identify all the relationships of a part, include or simulation run through time.
- It enables the generation of simulation iterations without the need to load in ANSA anything but the module to be modified.
- It offers a unique workspace for the planning of simulation needs at certain product development milestones.
- It offers the same view on the data through the pre- and post-processor. Post-processing actions can be triggered through ANSA and pre-processing actions directly through META.
- It embeds methodologies for model assembly that support all major assembly strategies used by the CAE community for inter-modular assembly, without imposing new, monolithic, assembly methods.
- It is applicable to all disciplines and it makes use of the common model concept, which enables the derivation of discipline-specific models from a single common model.
- It requires the minimum number of components: ANSA and META alone, or ANSA, META and SPDRM, for larger scale applications.
- The interfaces between the different components are available out-of-the-box.

1.1.3. About this document

This document will attempt to guide new users of the Modular Simulation Run Environment and provide some insight on the main concepts, the core tools used, as well as the inbuilt methodologies for Modular Run Management.

1.2. Enablers

There are 3 main enablers for the Modular Simulation Run Management:

1. The data management backend.
2. The workspace.
3. The methodology.

1.2.1. The data management backend

A key specification of the modular management environment is the possibility to manage different versions and variants of the include files. The data management solutions offered by BETA CAE Systems are employed in order to facilitate modular storage that features:

- Version control, on part, subsystem and system level.
- Management of model and loadcase variants.
- FE-representations management on part and subsystem level.
- Storage of key-results in association with the simulation runs.

BETA CAE Systems offers two alternative solutions for data management: The file-based ANSA Data Management (ANSA DM), which is suitable for use by smaller teams or as an entry-level data management solution, and SPDRM, which is a server-client solution. ANSA DM is embedded in ANSA which means that it's already available to all ANSA users. SPDRM is a program that requires a specific license and a different installation procedure.

The Modular Environment works equally well with both file-based ANSA DM and SPDRM. The ANSA/META user notices absolutely no difference when using any of the two alternative solutions.

1.2.2. The workspace

The *Model Browser* in ANSA is the main workspace for the management of models in a modular way. Through this tool the users can organize the model into modules, load and save modules, execute model assembly and loadcase set-up actions, run model checks and save the final simulation run, i.e. the main file to be submitted to the solver.

The DM Browser in ANSA and META is the workspace for the navigation of the data stored in the DM repository. Through this tool the users can search and find data, load them in the running session or export them in a directory on the disk, explore data relationships, compare different versions/variants of data, review simulation run results etc.

1.2.3. The methodology

The Modular Environment comes with built-in methodologies for the creation of modules and for the composition of modular models. The built-in processes can be enriched and can be adapted to the needs of each team with the aid of Python customization.

1.3. Terms

The main terms used throughout the Modular Environment are described in this paragraph.

1.3.1. Model organization

1.3.1.1. Simulation Run Architecture

The include files that are used in order to define a Simulation Run are organized in two groups, the **Simulation Model** and the **Loadcase**. The **Simulation Model** contains all the includes that represent sub-assemblies of the model in hand. The **Loadcase** contains all the information that relates to the loading scenario.

One level deeper, the **Simulation Model** may consist of one or more **Subsystems** and **Library Items** (which will be collectively referred to as **Modules** in this guide). The **Subsystems** represent sub-assemblies of the model and they evolve together with the design. **Library Items** at the Simulation Model level are usually additional files that contain assisting entities like material and property cards and have a separate lifecycle, different from that of the Subsystems.

In a similar fashion, the **Loadcase** may consist of **Subsystems** and **Library Items**, this time loadcase specific. Some examples of **Library Items** added to loadcases include the crash dummies and the barrier models, the files with control cards, the header files, etc. Adding **Subsystems** under the Loadcase may not be very usual but may be required in order to manage loadcase-specific set-up of vehicle seats and other loadcase-dependent subsystems like airbags, seat-belts, etc.

Loadcases may also contain **Header Builders** and **Target Points** Library Items.

Header Builders are special Library Items that are used in order to set-up Nastran and Abaqus headers based on the definition of boundary conditions and output requests.

Target Points are Library Items entities that are used in order to generate Simulation Runs based on a selection of Target Points and are mainly used for Pedestrian and Occupant protection loadcases.

One level deeper, the **Subsystem** consists of a collection of **Parts**. Usually, parts within a Subsystem are organized into a hierarchical structure with the aid of the Product Definition tree that is exported from PDM/PLM systems.

1.3.1.2. Model Browser Containers

The Model Browser Containers (or Model Containers or MBContainers) are the entities that are used in the Modular Environment in order to organize a Simulation Run with the architecture described above. More specifically, the Model Browser Containers are:

Name	Icon	ANSA entity type	Description
Part / Group		ANSAPART/ANSAGROUP	CAD Part and Group
Subsystem / Group Subsystem		ANSA_SUBSYSTEM	Model-specific sub-assembly
Library Item		ANSA_LIBRARY_ITEM	Library-related files
Simulation Model		ANSA_SIMULATION_MODEL	Model assembly
Loadcase		ANSA_LOADCASE	Loading scenario
Simulation Run		ANSA_SIMULATION_RUN	The Run. Combines a Simulation Model and a Loadcase.

All these entities are managed through their dedicated tab in the *Model Browser*.

Subsystems and Library Items are usually characterized as *base modules* in the sense that in most of the cases they are represented by a single file. However, Subsystem Groups, Simulation Models and Loadcases are characterized as *compound containers*, since they are compositions of base modules.

When a model is organized in a modular way, each model entity belongs to exactly one Model Browser Container. For example, a set can only be assigned to one Model Browser Container.

Through the *Model Browser*, it is possible to identify the entities contained in each Model Browser Container through the **Contents** bottom tab. At the same time, it is possible to identify the Model Browser Container to which an entity belongs through the **MBContainer** column in the *Database Browser* lists (and through script).

The *Model Browser* attempts to automatically organize the contents of each Model Browser Container into *virtual containers*. There are *virtual containers* for Geometry, Connections, Interfaces and Model Setup Entities:

- **Geometry**: This container collects the ANSAPARTs that have been assigned to the Model Browser Container, the part contents (i.e. geometric entities like faces, 3D Points, curves, etc., shell and solid elements) and their nodes, properties and materials.
- **Connections**: This container collects the Connections (spotwelds, adhesives, bolts, seamlines, etc.) and Connector Entities that are *internal* to the parts assigned to the Model Browser Container. This means that the contents of this container are dynamically maintained by the *Model Browser*, based on the connectivity information of Connections and Connectors. Note that the name of this container is adapted automatically depending on its contents and becomes **Connectors**, when it contains only Connector Entities and **Connections & Connectors** when it contains both Connections and Connector Entities.

  An exception to the “dynamic” addition/removal of Connections and Connectors in Subsystems is the behavior of Connecting Subsystems. Connecting Subsystems are a special type of Subsystems intended to hold intermodular connections, i.e. connections between different modules. Therefore, a Connecting Subsystem can be assigned connections manually, in which case the precondition for the connections to be *internal* to the Model Browser Container is not met.
- **Interfaces**: This container gathers all the entities that belong to the Model Browser Container but will finally be referenced by other entities, in higher level Model Browser Containers.
- **Model Setup Entities**: This container shows all the entities that have been manually dropped on the Model Browser Container and do not fall in any of the categories above (e.g. any element except for shells and solids, sets, contacts, hourglass cards, cross sections, time history markers, etc.). Furthermore, any Generic Entity Builder dropped on a Model Container will also end-up in this container.

  The contents of this container are not maintained automatically. However, it is possible to detect the relevance of Model Setup Entities with a Model Browser Container with the aid of the contents check described in paragraph 2.2.1. With this check it is possible to identify entities that should belong to a container but currently don’t, based on the geometry assigned to the container. Detection of the opposite scenario is also possible: Model Setup Entities that currently belong to a container but seem to be unrelated to its geometry.

More information on Model Organization is given in Chapter 2.

1.3.2. Adapters

An important specification for the Modular Environment is the possibility to reuse the same container under different Simulation Models and Loadcases. For example, it must be possible to reuse the same Subsystem (e.g. a wheel) in different locations under the same Simulation Model or reuse the same barrier Library Item in different positions under different Loadcases. In order to achieve this, the information that has to do with the *use* of a base Model Browser Container within the context of its parent, compound Model Browser Container, cannot be part of

the Model Browser Container itself, but must be kept by the *consumer*. This need is covered by the *adapters*, which hold all the information regarding the use of a Model Browser Container within the context of its parent container.

There is one *adapter* for each *instance* of a Model Browser Container within a compound Model Browser Container. The following ANSA types represent *adapters*:

ANSA entity type	Description
ANSA_SUBSYSTEM_GROUP_ADAPTER	Binds a Subsystem with a particular Subsystem Group
ANSA_SIM_MODEL_ADAPTER	Binds a Subsystem or a Library Item with a particular Simulation Model
ANSA_LOADCASE_ADAPTER	Binds a Subsystem or a Library Item with a particular Loadcase
ANSA_SIM_RUN_ADAPTER	Binds a Library Item, Simulation Model or Loadcase with a particular Simulation Run

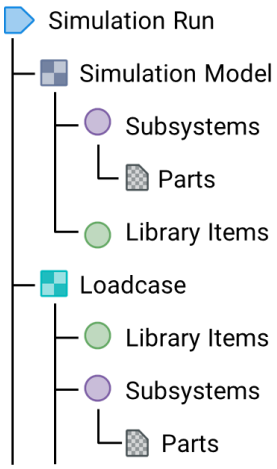
The adaptation information is usually expressed in attributes. Characteristic examples of adapting attributes are the target id ranges, the position transformation information, etc.

The topic of “adaptation” is covered exhaustively in Chapter 5.

1.3.3. Includes

In the Modular Run Environment, includes (i.e. include entities) are only used as an *output medium*, i.e. as a means in order to represent the different modules by different files on the disk and do not need to be managed by the user.

Usually, the Simulation Run is a nested structure of -at least- three levels:



In the Modular Environment it is possible to represent any of these levels by an include file, creating multi-level nesting between different includes. When a Model Browser Container contains some other Model Browser Containers, it is possible for the former to be written out as a **monolithic** file, containing the contents of the child Model Browser Containers directly in its file, or alternatively as a file **with references**, which corresponds to a file with *include keywords* to the include files of all child Model Browser Containers. The exact file structure of the Simulation Run is controlled through the Save in DM settings of the ANSA.defaults (more info in paragraph 3.3).

For Subsystems and Library Items in particular, it is technically possible to organize their contents into more than one include files. In this case, the main keyword file of the Subsystem or Library Item will contain *include keywords* to its sub-includes. However, in a structure like this, the individual sub-includes are always bundled together with the main keyword file of the Subsystem or Library Item and cannot be managed individually.

1.3.4. Simulation Configuration Tables

The *Simulation Configuration Tables* are template files that describe the composition of compound Model Browser Containers. These templates are saved in DM as Library Items. There can be 3 types of *Simulation Configuration Tables*:

1. **Simulation Model Configuration Tables**: These templates contain all the Simulation Models that must be built for a particular development milestone and the Subsystems they contain. Essentially, they are used to capture the different model variants of the model. The different Simulation Models may consist of Subsystems with different Variants (e.g. left hand drive / right hand drive BiW, standard roof / panorama roof, v6 petrol / hybrid engine, etc.) or Subsystems with different Representations (e.g. full FE / lumped mass representation of doors, use / don't use representation of Subsystems).
2. **Loadcase Configuration Tables**: These templates contain all the Loadcases that must be built for a particular Discipline or even for several Disciplines and their contents, that can either be Library Items or Subsystems.
3. **Simulation Run Configuration Tables**: These templates contain all the Simulation Runs that must be simulated for a particular development milestone, for a particular Discipline, or even for several Disciplines. Essentially, they are used to capture the combination of Simulation Model variants with Loadcases.

1.3.5. Assembly and Loadcase set-up

1.3.5.1. Interface entities

With the term *Interface Entities* we refer to all those entities that belong to a Subsystem or Library Item but will be used for its integration to the assembly or for the set-up of the loadcase. These entities are marked in a distinctive way, so that they are *exposed* by their Model Browser Container for use by higher level containers.

There are four different types of interfaces, each with its own marking method:

1. **Interface nodes**: These are nodes that are used for inter-modular assembly or loadcase set-up. These nodes are marked by special entities, the so called Assembly Points (A_POINTs) and Loadcase Points (LC_POINTs) that are used for assembly and loadcase set-up respectively.

   There are several cases where the same node marked for assembly also needs to be marked for load-case set-up. In the Modular Run Environment it is possible to deal with this requirement in two alternative ways:

   a. By marking the same node with both an Assembly Point and a Loadcase Point.
   b. By using Assembly Points for both assembly and loadcase setup.
2. **Interface sets**: These are sets that are used for the definition of cross-modular entities, like for instance global contacts, curb weight, time history output, boundary conditions, etc. These sets are marked as interfaces through the setting INTERFACE=YES within the set card.
3. **Interface properties**: These are properties that are used for inter-modular assembly. These properties are marked as interfaces through the setting INTERFACE=YES within the property card. The most common example is that of the rigid patches for LS-DYNA, where rigid properties that reside within each subsystem are referenced by rigid bodies stored in another Subsystem or in the Simulation Model.
4. **B.C. SETs**: These interfaces are the Boundary Condition Sets used by boundary conditions in Nastran deck. Every time an SPC, MPC, NSM, DMIG, etc. is created within a Subsystem or Library Item, the Boundary Condition Set Id it uses is automatically considered as an *interface entity*.

1.3.5.2. Smart Assembly

The term Smart Assembly refers to the methodology used for includes' assembly in the Modular Run Environment and covers the processes of creation and update of connecting elements and connecting includes. In contrast to conventional includes' assembly processes, Smart Assembly:

1. Doesn't require the loading of all the individual includes in order to create the connecting elements, which is very time consuming, especially when talking about modules with few hundred thousands or even millions of elements each
2. Doesn't rely on the reference of "frozen" node ids and PIDs from the connecting elements, which is very error-prone and requires very disciplined id management on module level

The *Smart Assembly* distributes the effort for the creation of the first intermodular assembly equally between the model building teams, that prepare the individual modules, and the "model integrator", who brings the modules together and creates the connections between them. The contribution of the model building teams is limited to the marking of the interface locations on module level with the methods described in the previous paragraph. Then, the "model integrator" has to load the interface entities, create connecting entities and realize them, in order to create the first version of the connecting subsystems.

From that point on, the update and verification of the connecting subsystem after modifications of the individual modules is managed automatically by the system.

Note that the term *Smart Assembly* is not limited to model assembly and may also refer to the gathering of interface information for the definition of Loadcase entities. An example is the use of Loadcase Points defined in Subsystems for the definition of collective boundary conditions like loads, constraints or output requests.

1.3.5.3. Assembly Sets

Assembly Sets are used in inter-modular assembly in order to bring together individual interface sets that come from different modules. Such "merged" sets are usually used for composition purposes, e.g. for the definition of global contacts that contain properties contributed by all modules of the model, for the creation of sets used for mass balancing, etc. Assembly Sets are populated with the aid of *Domain Finders*, ANSA Generic Entities that can search and find interface sets within Model Browser Containers.

Assembly related topics are covered exhaustively in Chapter 6.

1.3.6. Data repository

1.3.6.1. DM Objects

A *DM Object* (or *DM Item*) is a data management entry that represents a Model Browser Container that was saved in the data repository. There are DM Objects for Parts, Subsystems, Simulation Models, Loadcase Templates, Simulation Runs, Library Items, etc. A DM Object is comprised of one or more files and certain metadata.

1.3.6.2. Metadata

The term *metadata* refers to data that are used to describe other data. In the Modular Environment, *metadata* is used in order to reveal more elaborate information about the object in hand. For example, the *Module Id*, the *Project*, the *CAD Release* and the *Variant* of a Subsystem are among the metadata that needs to be filled before saving the Subsystem in the data repository.

At the same time, there are other attributes that are given to the DM Object automatically during Save, like for example the *Iteration*, which represents its study version and the *DM Creation Date*, which stands for the time the DM Object was created in the database.

Finally, there's another set of additional attributes that characterize the content of the file, so that one can get an idea of what's contained in the file without having to open it. Such attributes may include the Subsystem mass, its inertia tensor, its center of gravity and orthogonal bounding box dimensions, or even more specialized information, meant to be used during assembly, like for instance the id ranges of the entities it contains or the names and locations of its interface points.

One could also consider the tree structure (hierarchy) as a particular type of *metadata*. For a Subsystem, its tree structure would reveal the parts and groups it consists of in a hierarchical manner. For a Simulation Model, its tree structure would list the Subsystems and Subsystem Groups it consists of.

Data Management related topics are covered exhaustively in Chapter 3.

1.3.7. The "Build" Process

1.3.7.1. Definition file

Each Model Browser Container that is saved in DM in solver keyword format, comes with a source definition file, in ANSA format, that is used as a reference for making modifications to the solver file. This file is either saved as an attachment to the DM object of the Model Browser Container with Solver File Type, under the attribute *Definition ANSA File*, or as a separate DM object with File Type ANSA. So, depending on Model Browser Container type, the default behaviour is:

- For Subsystems: The user creates and saves Subsystems in ANSA format (i.e. as DM objects with File Type ANSA). The moment the solver keyword file of a Subsystem is created from its ANSA DM Object, it is stored as a new DM object with Solver File Type and a link is automatically created between the source and product DM Objects.
- For Subsystem Groups, Simulation Models, Loadcases and Simulation Runs: The definition file is saved as an attachment to the DM object with Solver File Type.

1.3.7.2. Build process

Each Model Browser Container has a process that must be followed in order to verify that it's ready to be saved in the data repository and it meets the requirements of its parent container for the composition of the model. Theoretically, executing the *Build Process* of a Model Browser Container on its Definition ANSA File, the product of the process is ready to be saved in DM as a solver keyword file.

The *Build Process* consists of several individual steps, the *Build Actions*. Each Model Browser Container comes with a default *Build Process* that can be extended and enhanced with the aid of scripting, by adding one or more custom actions.

During the execution of a *Build Process*, its actions are executed from top to bottom. Each action has a status that will either be undefined (grey) before it's executed, or OK (green) in case it is executed successfully or Error (red) in case its execution fails. If a Build Action fails, the Build Process is halted with Error Build Status for the Model Browser Container. When all Build Actions are executed successfully, the Model Browser Container gets Build Status OK.

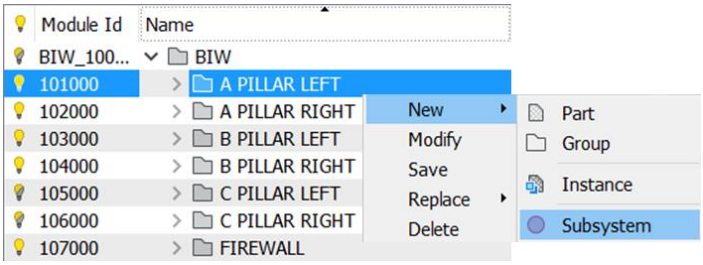
More information on the Build Process is given in Chapter 4.

2. Organizing a model in the Modular Environment

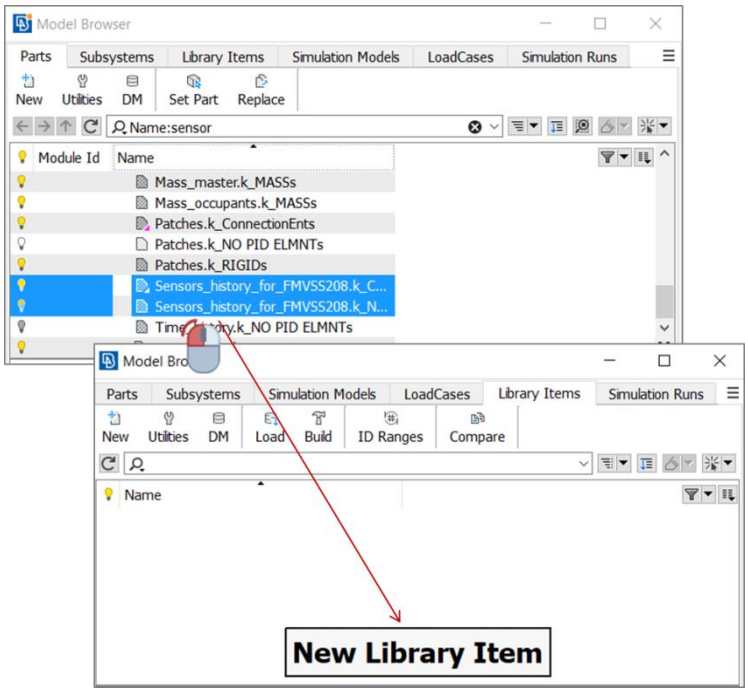
2.1. Adding entities in Model Browser Containers

Depending on the nature of each module, the source data for its creation may either come from a part structure, possibly created through the “CAD to ANSA” process, or from an existing include file. Both cases are covered in the Modular Environment.

2.1.1. Creating a Module from a selection of Parts/Groups



In order to create a Subsystem from a part structure, the user can select any Parts/Groups from the Parts tab of the Model Browser and use the context menu option **New>Subsystem**. A new Subsystem will be created with the name of the selected Parts/Groups. However, this is only true with A) the default schema, B) if a single part/group is selected or C) choosing the option “Create a Subsystem for each Part” in case multiple parts/groups are selected. At the same time, an Include will also be created and linked with the Subsystem.

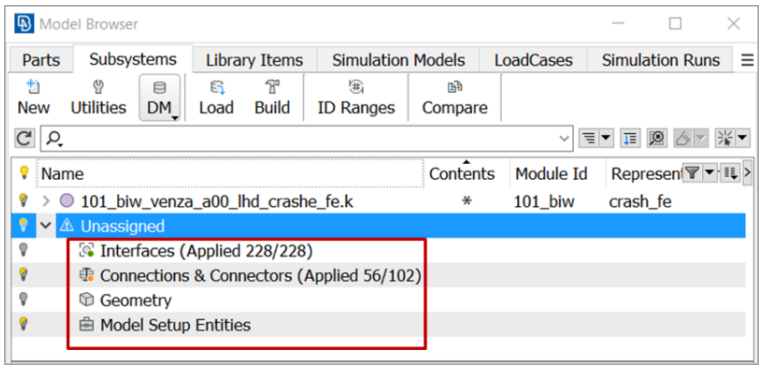


In order to create a Library Item from a part structure, the user can select any Parts/Groups from the Parts tab of the Model Browser and drag and drop them to the Library Items list.

A new Library Item will be created. At the same time, an Include will also be created and linked with the Library Item.

ANSA will collect all entities contained to the selected Parts/Groups and distribute them to the **Geometry**, **Connections**, **Interfaces** and **Model Setup Entities** virtual containers.

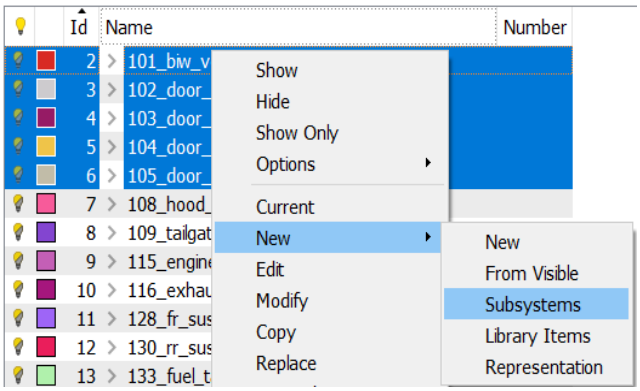
- **Geometry**: This container collects the ANSAPARTs that have been assigned to the Model Browser Container, the part contents, their nodes, properties and materials.
- **Connections**: This container collects the Connections and Connector Entities that are *internal* to the parts assigned to the Model Browser Container.
- **Interfaces**: This container collects any Assembly and Loadcase Points that are internal to the parts assigned to the Model Browser Container.
- **Model Setup Entities**: This container shows the rest of the entities that do not fall in any of the above (e.g. any element except for shells and solids, sets, contacts, cross sections, time history markers, etc.).



If there are entities in the model that are not assigned to any Model Browser Container, they will be displayed under the **Unassigned** container, organized to virtual containers according to their type. In case unassigned entities are not present in the model, the respective container will not be visible.

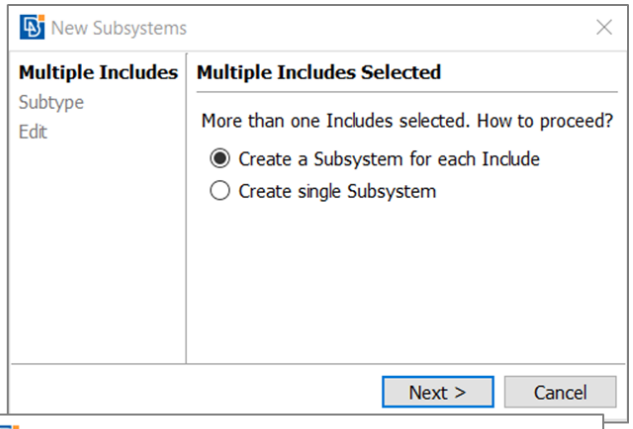
Note ! It is possible to remove an entity from its Model Container by dropping it on the Unassigned container.

2.1.2. Creating a Module from a selection of Includes



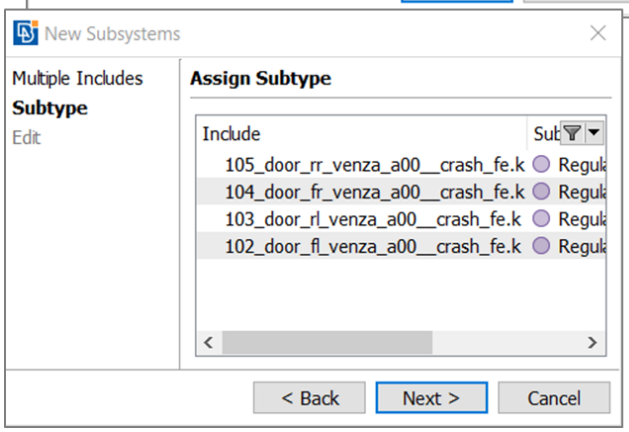
A module can be created from a selection of Includes in the following ways:

1. From the Includes list through the context menu options New > Subsystems/Library Items

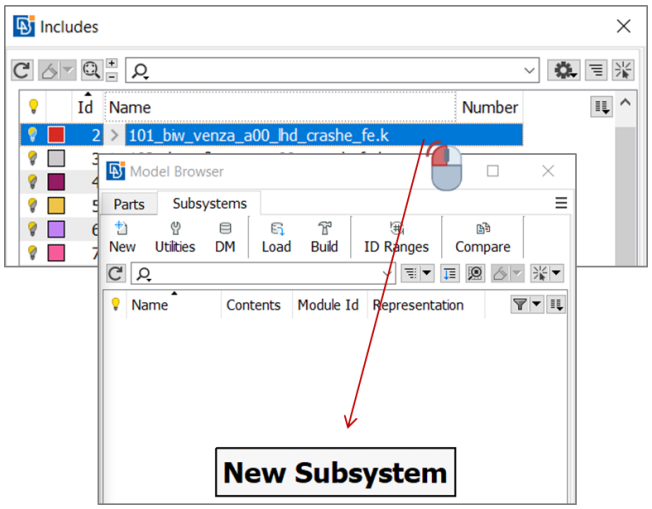


A wizard is displayed that allows the user to control parameters of the process.

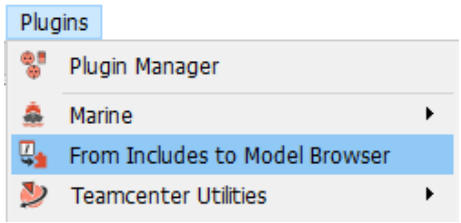
In case multiple includes were selected for Subsystem creation, the user is prompted to specify if a single Subsystem will be created containing all the selected includes or if a Subsystem will be created for each Include.



Also, it is possible to define which includes are going to be converted to connecting and which to regular subsystems.



2. With drag and drop Includes on an existing, empty, Subsystem or Library Item in Model Browser or with drag and drop Includes in the empty space in Subsystem and Library Items lists in the Model Browser.



3. Another option is the “From Includes to Model Browser” plugin. This is a built-in migration tool that can be used to convert models organized with includes to models organized with Model Browser Containers. It can be accessed through the respective menu bar option.

2.1.3. Creating a Module from a selection of entities in the Database Browser

It is possible to create a module (Subsystem or Library Item) with drag and drop of entities from the *Database Browser*. The behaviour on drop is slightly different depending on the types of entities dragged:

Dragging shells and solids

On drop of an arbitrary selection of shells and solids on a new module, only the entities that form complete parts are actually dropped together with their parts. If the user wishes to drop only the selected elements then the ANSA Parts must be split manually beforehand.

Dragging other entities

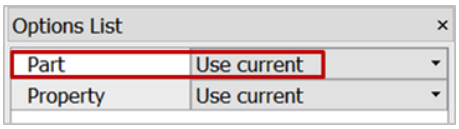
In case the user drops on a module some elements that belong to an ANSA Part that contains more entities than the selected, ANSA will automatically split the existing part, creating a new part containing the selected elements only and will add this new Part to the module.

Dropping on the module any other entities that are not assigned to ANSA Parts (e.g. sets) will assign them to the module directly.

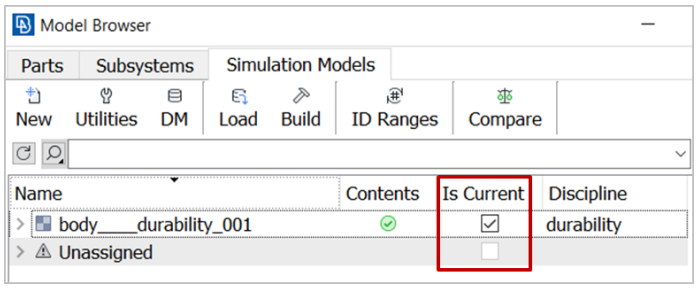
2.1.4. Control the Model Browser Container of newly created entities

Despite the fact that ANSA offers automated functionality that distributes entities in the respective Model Browser containers, there are cases where the user needs to control to which container newly created entities will be placed. In such cases, the user can define the desired destination using the *Part* setting in the *Options List* window of the currently active functionality (when applicable). The relevant options are two:

Part: Use current



This setting will place the newly created entities in the Model Container set as *current*.



There are cases where the user has to select if the newly created entities will end up in a specific Subsystem or a higher level Model Browser container. Thus, the user can specify any of the existing Model Browser containers as current from its context menu current using ***Mark > Current > On*** action. However, if those entities are associated with another Part (which is not the current), then they would be assigned to that Part/Subsystem. Furthermore, the current container of the model can be easily identified using the "*Is current*" attribute in the Model Browser. Another way to identify the current container is from the top area of the graphics window which is displayed.

Note! Only one of the Model browser containers of the model can be set as current at a time.

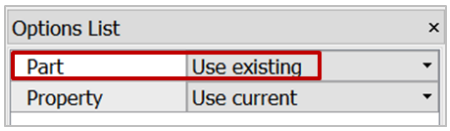
The ANSA.defaults setting that activates this mode is *New element Part* which is set by default to Use existing, but Use current is the recommended option when using the modular run management approach.

Part: Use existing

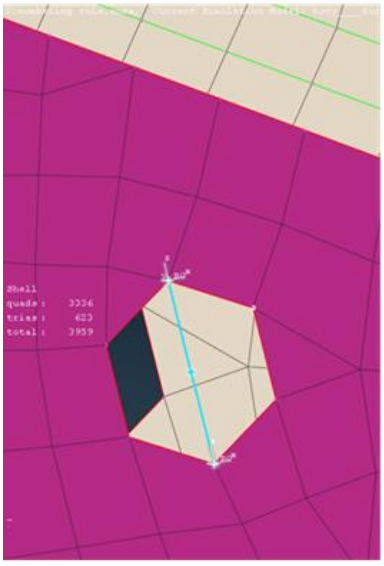
This setting will place the newly created entity in the Model Container of the selected entities. More specifically:

- If the selected entities belong to the same Model Container, then the newly created entities will be assigned to this Model Container.
- If the selected entities belong to different Model Containers, then the newly created entities will be assigned to their common parent -if it exists. For example, if the selected entities belong to 2 Subsystems, the newly created entities will be automatically assigned to the Simulation Model that contains both Subsystems. If no common parent exists, then the user is prompted to decide the target Model Container.

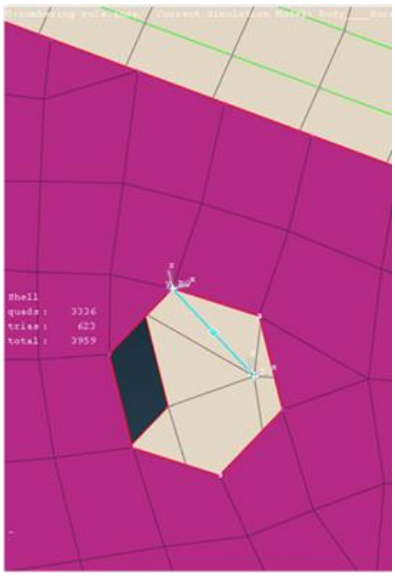
The ANSA.defaults setting that activates this mode is *New element Part* which is enabled by default.



Let's take as an example the creation of a beam element between two nodes that belong to the same Subsystem and two nodes that belong to different Subsystems.



Nodes belong to the Same Subsystem



Nodes belong different Subsystems

By selecting the Use existing option the newly created CBEAM element will be placed in the Subsystem that the two selected nodes belong. While in the following case where the each node belongs to a different Subsystem the created element will be placed under the Simulation Model, since it is their common parent container.

2.2. Self-containment of Model Browser Containers

A Model Browser Container could be characterized as “self-contained” if includes entities which do not reference entities of other modules. A common need when working with modular assemblies is to be able to verify that no entity that should belong to a module has been accidentally assigned to some other module. ANSA is able to identify the proper "destination" of entities based on what they are attached to. Taking for example a contact definition that uses two sets, the contact could be internal to a Subsystem if both sets only contain entities that belong to that Subsystem.

Keeping the modules self-contained brings certain advantages:

- It eases the “good housekeeping” of modular models. Users are able to work on individual modules, without missing entities defined in other modules
- It eliminates the need for monitoring the interdependencies between modules

Self-contained modules may ease the overall model handling but, in reality, they are not as common. We have 3 kinds of *known* deviations from self-containment:

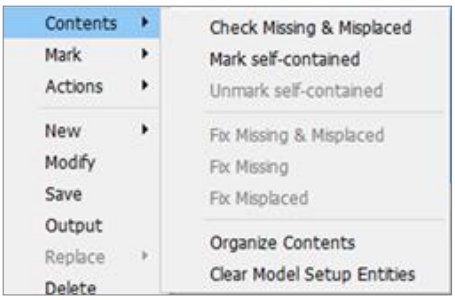
- Sharing of entities like properties and materials between several modules with the use of property or material databases
- Sharing of nodes/properties/sets in case of Connecting Subsystems (Connecting Subsystems are not self-contained “by definition”, since their elements/connectors reference nodes/properties/sets that belong to the connected subsystems)
- Sharing of nodes/sets in case of loadcase entities (a loadcase is not self-contained “by definition”, since the BCs reference nodes/sets that belong to the Subsystems)

Having modules that are not self-contained is perfectly valid in the Modular Run Environment. However, it’s always useful to be aware of deviations from self-containment and make sure they are deliberate.

2.2.1. Check self-containment of Model Browser Containers

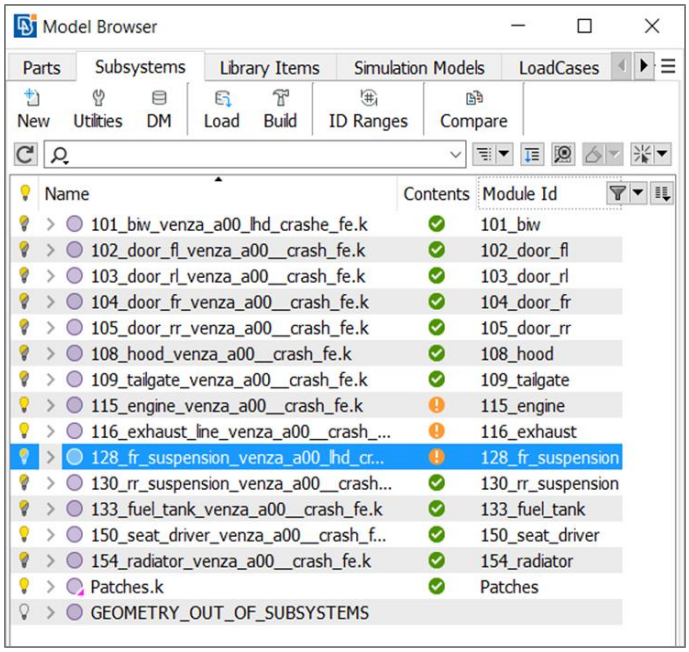
To assist the user in checking whether several modules loaded in an ANSA session are self-contained and in case they aren’t, verify that any deviations are known and deliberate, special functionality is available that makes it possible to identify for a Model Browser Container:

- **Missing** entities, i.e. entities that should belong to this container but currently do not
- **Misplaced** entities, i.e. entities that belong to this container while they should not.



After identifying any Missing or Misplaced entities the problem can be resolved by either adding the missing entities to the Subsystem or by removing the misplaced entities.

This functionality can be found in the context menu of all Model Containers (except for Runs), under the option **Contents**.

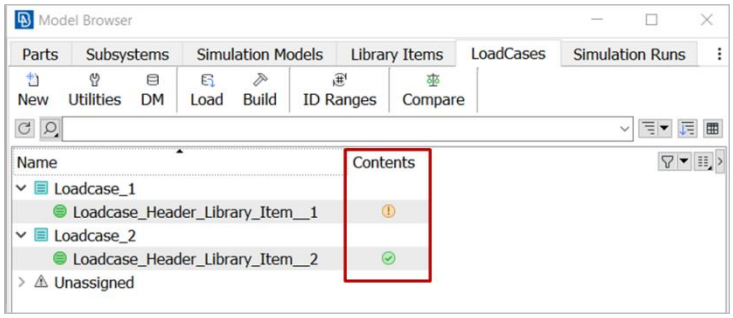


In order to get started, the user can select several Subsystems and run the function **Contents > Check Missing & Misplaced**. In the **Contents State** column of the *Model Browser* lists, an indication of the contents status is shown ("green" is used to mark modules that were identified as self-contained and "orange" for modules that have missing or misplaced entities).For Subsystems that are not self-contained, the user can get more information on the deviating entities through the **Contents** tab.

In order to fix deviations, the user can press the **Fix** button in the **Contents** tab or activate the functions **Contents > Fix Missing/Fix Misplaced/Fix Missing & Misplaced** of the context menu. Alternatively the user can fix the reported deviations selectively by picking the entities that should be added or removed from the module among all the entities identified automatically.

This operation can be handled for several Model Browser Containers in a single shot with the aid of **the Contents > Organize Contents** function of the context menu. This function checks for self-containment and fixes any deviations automatically. However, it is recommended to use this function only at early stages of the model organization.

The same principles can be also applied in Loadcase Header items. During the set-up of the Header due to user prone errors deviations may occur between the Include contents of the Loadcase Header and the actual entities referenced in the Loadcase Assistant in the Model Browser. Similarly, to what is described above Check Missing & Misplaced functionality is available to identify such incidents.



For Loadcase Headers which are not self-contained, the user can get more information on the deviating entities through the **Contents** tab. Additionally, **Contents > Fix Missing/Fix Misplaced/Fix Missing & Misplaced** functionalities or **Contents > Organize Contents** function of the context menu can be utilized in order to fix any deviations automatically.

Few common questions on the organization of module contents are answered below:

What if an entity is claimed by 2 or more Subsystems?

In case an entity is claimed by 2 or more Subsystems, it is an indication that it may need to be assigned to a Connecting Subsystem or to the higher level model container, e.g to the Simulation Model.

What if an entity that is assigned to the Simulation Model is claimed by a Subsystem?

If an entity that belongs to the Simulation Model is claimed by a Subsystem, i.e. it is reported as missing from the Subsystem and as misplaced for the Simulation Model, the **Fix Missing** or **Fix Misplaced** action on the Subsystem and on the Simulation Model respectively will fail with a message: *'Entity was detected under other Model Browser*

Containers.' In cases like these the user should manually fix the problem by moving any missing entities to the Subsystem with drag and drop.

What if an entity that belongs to the Subsystem is claimed by the Simulation Model?

This error cannot occur. Entities that should belong to the Simulation Model because they are used by more than one Subsystems but are assigned to a particular Subsystem are still considered to be parts of the Simulation Model. As a result, this situation is not reported as an error on the Simulation Model level. Of course, running the contents check on the Subsystem that contains such entities will report them as misplaced, in the sense that they do not relate exclusively to this Subsystem.

Which entity types can be rearranged in this fashion? Can I rearrange nodes?

Entities that can be rearranged using this methodology are Connections, Connectors, Interfaces and Model Setup Entities, i.e. everything except for geometric entities, shell and solid elements and ANSA parts.

Can this functionality work on any Subsystem?

This functionality only works for Subsystems that contain Geometry/FE shells or Solids (not shells/solids that are FE representations of Connections). In case a Subsystem doesn't contain such entities then contents relations cannot be detected and the user should manually place the proper entities in the Subsystem and mark it as self-contained.

2.2.2. Protect self-containment during modular I/O

During keyword file input and output, the integrity of non self-contained models is maintained with the aid of the DEFINED marking. Entities like nodes, properties, materials, sets, load-curves etc. (i.e. that are referenced across different modules) have a DEFINED flag that can get the value YES or NO:

During input of a module, missing entities are automatically created and are marked as DEFINED=NO. This is done in order to maintain the model integrity (i.e. keep the proper ids referenced in cards) but at the same time ensure that these new created entities are not exported on output.

During output of a module, all the entities that are referenced by its contents but are not contained in it, will not be exported. Subsequently, during input of this module, these entities will be created automatically and will be marked as DEFINED=NO.

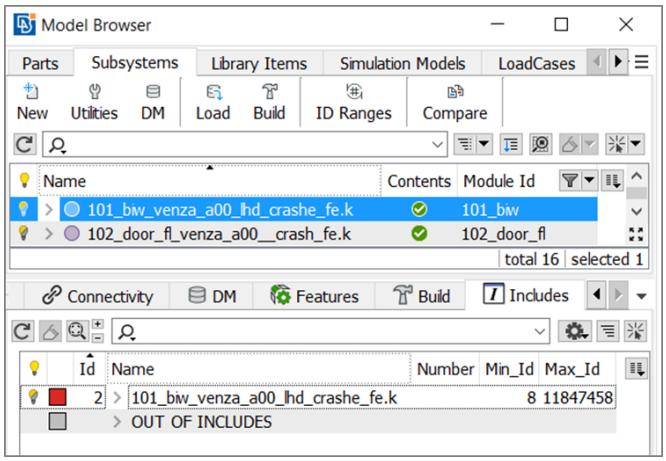
The same idea, of marking non-contained entities as undefined, is also applied when saving the modules in ANSA format. Entities that are referenced by the contents of the module being saved, will automatically be set to DEFINED=NO in the created ANSA file.

2.3. Association of Model Browser Containers with Include entities

Each Model Browser container is associated with an include entity. However, the include is not created the moment the Model Browser container is created. Instead, it is created the moment some entities are dropped to the Model Browser container or when a numbering rule is set to it, even if it's still empty.

Once a Model Browser container is associated with an include, it controls the name of the include entity. Any change in the name of the Model Browser container is instantly reflected to the name of the Include entity.

The include entity associated with a Model Browser container can be seen in the **Includes** bottom tab of the *Model Browser*.



The management of the contents of Model Browser containers must be done solely through the *Model Browser*. Dropping entities in the underlying includes is generally not recommended and will lead all self-containing algorithms (Missing-Misplaced-Organize contents) to error.

2.4. Using Subsystem Groups

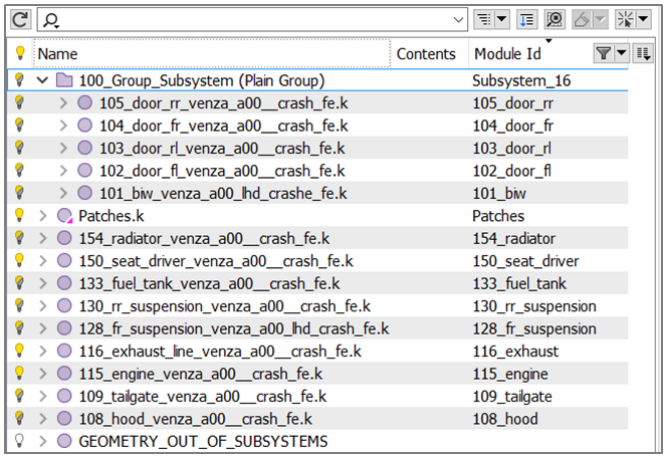
The model assembly can be subdivided into smaller units with the use of Subsystem Groups. Subsystem Groups enable the grouping of Subsystems and their management either as monolithic files or, more often, as files with references to the individual Subsystems. Groups offer great flexibility in organizing a Simulation Model according to the needs of different teams and disciplines. They can hold their own Connections and Model Setup Entities, and, in essence, they create an additional level of organization between the Subsystem and the Simulation Model.

For example, it is possible to create a Simulation Model that contains individual monolithic Subsystems like the body-in-white, the closures, etc., to facilitate the needs of NVH teams and, at the same time, create a Simulation Model where the body-in-white is represented by a Group Subsystem and is broken down to several Subsystems like underbody, top-hat, front module, rear module, to facilitate the higher granularity usually required by the crash teams.

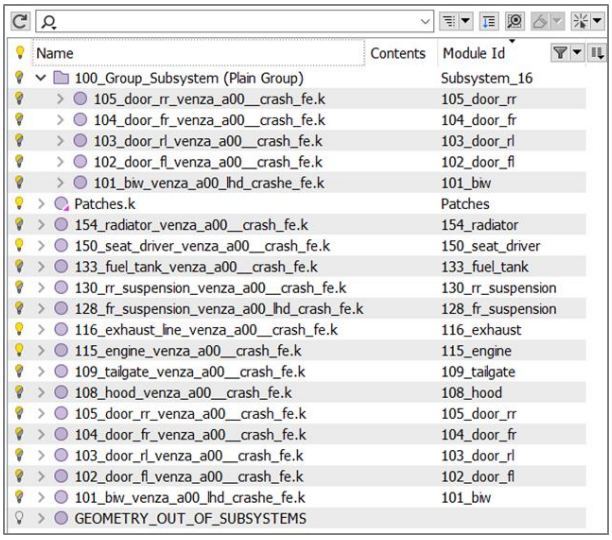
Selecting the option **New > More... > Group** from the context menu in the **Subsystems** tab, the user can create a new empty Subsystems Group.

To add entities to the newly created Group, the user can select one or more Subsystems from the **Subsystems** tab and drag and drop them onto the newly created Group. Alternatively, a Subsystems Group can be created directly from a selection of subsystems, by activating the context menu from the selection and using the option **New > More... > Group**.

Note! Subsystems that belong to Subsystem Groups are not visible at the top level of the Subsystems list when using the default “Tree View”. However, if the view is changed to “Tree View (show unadapted)” then the Subsystems that belong to Subsystem Groups become visible at the top level in the Subsystem list.



Tree View (Default)



Tree View (show unadapted)

Subsystem groups can be also used in order to facilitate the configuration of Simulation Models.

When defining variants of Simulation Models, there are “families” of Subsystems whose members cannot participate in a Simulation Model simultaneously. For example all the alternative engine variants of a vehicle model belong to the “engine family”. Only one engine Subsystem should be used in a Simulation Model at a time. Similarly, there are other groups of Subsystems whose members should always go together. For example, an engine variant may be combined with a particular transmission variant. Using this engine variant should automatically activate the corresponding transmission variant too.

Such relationships can be described with the aid of special types of Subsystem Groups. In this case, the Subsystem Groups need to be marked as either **exclusive** or **inclusive**, in order to denote the desired relationship among their contents. These relationships are finally visualized in the table view of Simulation Models.

Inclusive groups: All contained Subsystems should take part in a Simulation Model at the same time. Removing one, all of them will be removed.

Exclusive groups: Just one of the contained Subsystems can be part of a specific Simulation model at a time. Activating another Subsystem that belongs to the group will automatically uncheck the previously active Subsystem.

Note that a Subsystem Group, by default, is neither inclusive nor exclusive and is considered to be a *plain* group. The type and behaviour of a Subsystem Group can be set through the context menu options **Mark > Inclusive / Exclusive / Plain Group** .

Subsystem Groups share the same entity type with regular Subsystems, i.e. ANSA_SUBSYSTEM. In order to distinguish a Subsystem Group from a standard Subsystem, one can refer to the default attribute “Is Group”:

Model Browser Container	Entity Type	Is Group
Subsystem	ANSA_SUBSYSTEM	NO
Subsystem Group	ANSA_SUBSYSTEM	Plain / Inclusive / Exclusive

Adapting attributes

As already mentioned in Chapter 1.3, it is possible to reuse the same container under different Simulation Models and Loadcases with the use of the adapting attributes. These attributes hold all the information regarding the use of a Model Browser Container within the context of its compound container.

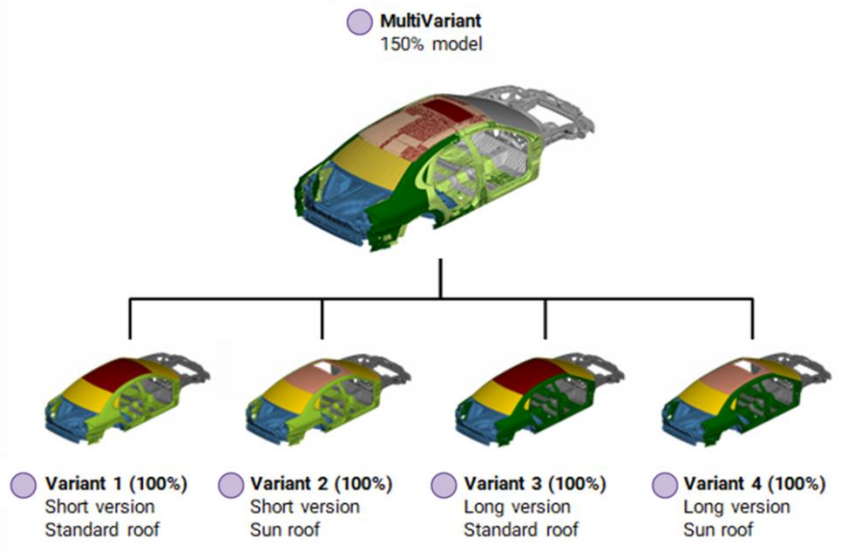
Similarly with the use of a Subsystem under different Simulation Models, the same Subsystem can be also included in different Subsystem Groups or into the same Subsystem Group more than once, in different positions and with different id ranges.

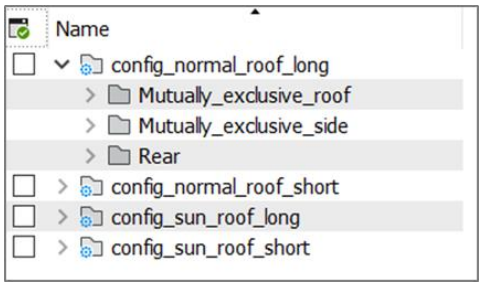
For more information refer to Chapter 5, Adaptation.

2.5. Handling 150% Subsystems

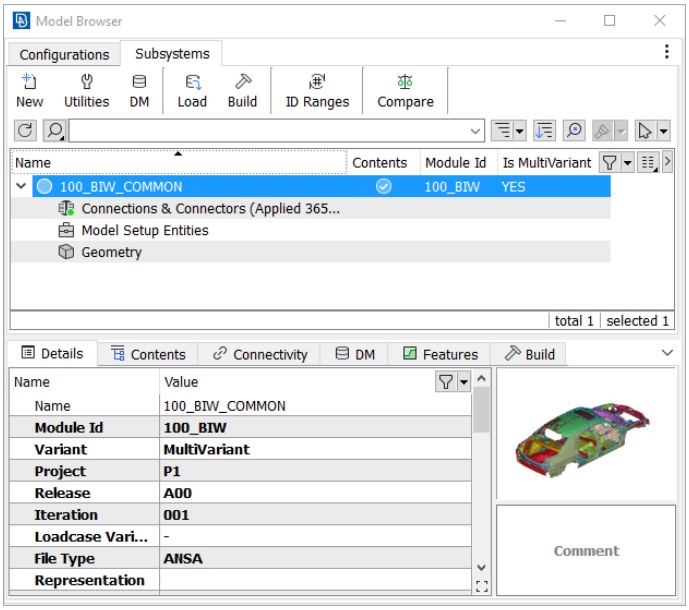
In the Modular Environment, in order to efficiently handle different variants of Subsystems, it is possible to couple Subsystems with Part Configurations. With this setup, an ANSA Subsystem can hold a reference to the 150% structure of the model and the Part Configurations associated with it, and then it be used as the source for the creation of 100% Subsystem variants.

In the example below, 4 different variants of the body-in-white Subsystem can be created from a single 150% ANSA model.



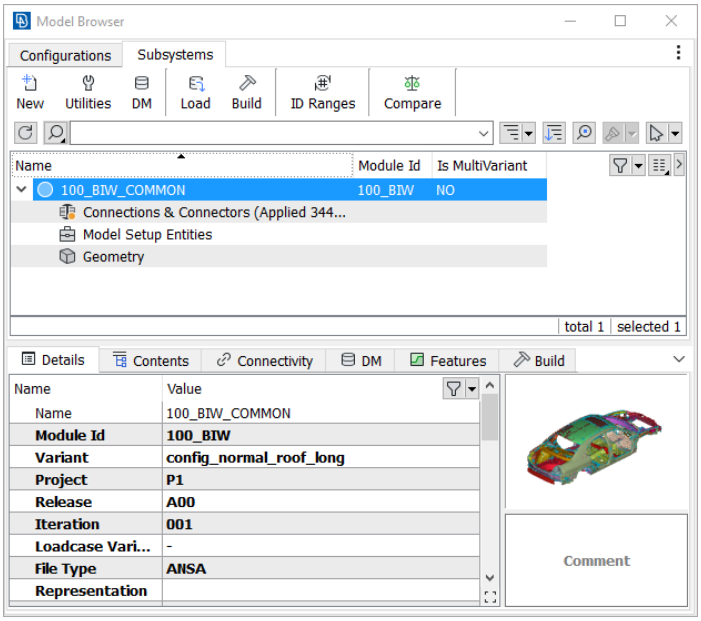


In order to manage a 150% Subsystem in this way, the user first needs to create the 4 different Part Configurations in the **Configurations** tab of the *Model Browser*.



In the next step, and while no configuration is active, the top-level group(s) of the part structure are added to a Subsystem. The Subsystem detects that the part structure is associated with inactive Part Configurations and sets automatically the "Is MultiVariant" field to "YES".

When Saving in DM a **MultiVariant** Subsystem, all configurations are saved along.



After activating any configuration in the **Configurations** tab, the "Is MultiVariant" field is set to "NO" and the variant attribute of the Subsystem gets automatically the name of the active part configuration. In case that more than one Subsystem is present, ANSA can recognize the Subsystems that are involved in the active part configuration.

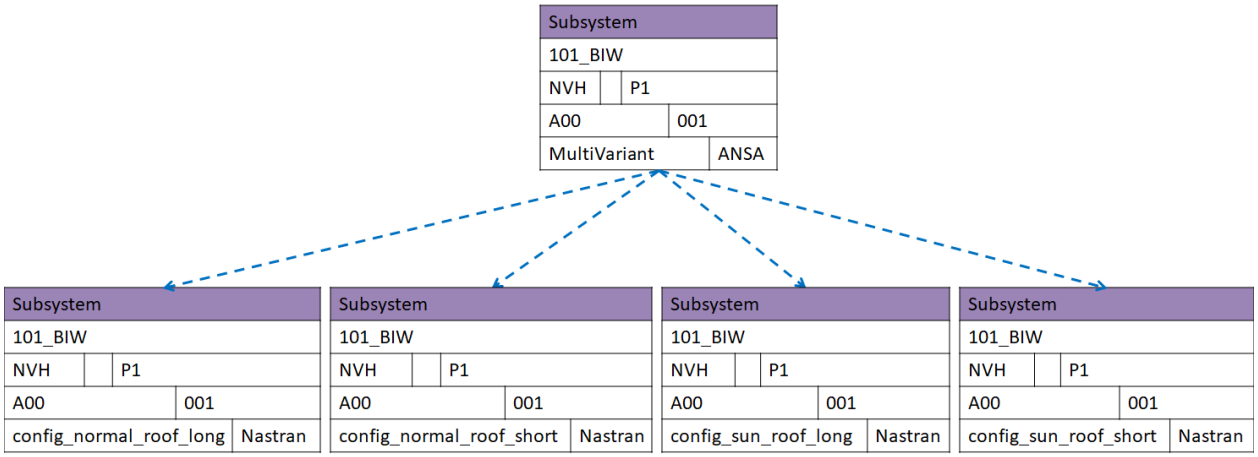
Note that the default variant attribute is "DM/Variant". However, it is possible to specify a different variant attribute through ANSA.defaults:

Keyword(As written in file)	Keyword(As displayed	Current value	Start-up value
variant_attribute			
DM Parameters	DM Parameters		
Variants Settings	Variants Settings		
variant_attribute	Variant Attribute	DM/Variant	DM/Variant

Upon saving a Subsystem in DM while a configuration is active, only the part structure and contents that relate to the active configuration will be saved along.

The **MultiVariant** Subsystem is meant to be the reference file that is used to generate the different 100% Subsystems. Ideally, a 100% Subsystem should report as its "Definition File" the 150% **MultiVariant** Subsystem.

The graph below shows the relationships between the Definition File and the resulting Subsystems for the body-in-white example shown above.



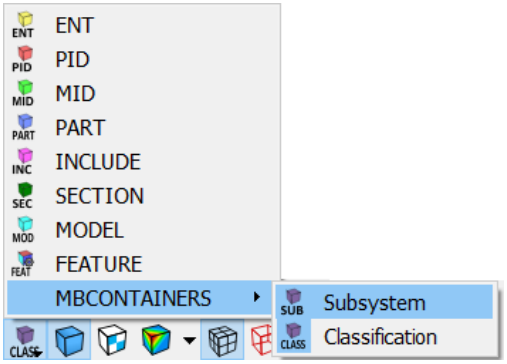
Note ! ANSA does not automatically create a "generation" link between the 150% ANSA file and the resulting 100% solver Subsystems.

The same concept also applies to Connecting Subsystems.

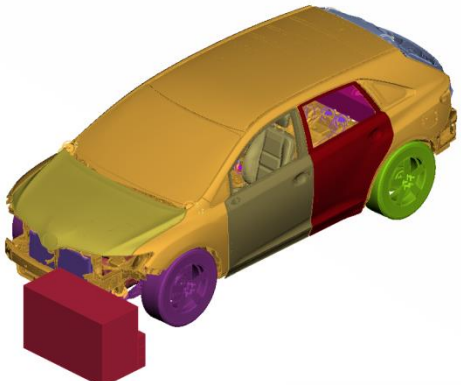
A 150% Connecting Subsystem can be utilized to manage the intermodular connections that will be used by different Simulation Model Variants. More information on this approach is given in paragraph 6.4.4.

2.6. Reviewing modularization of models

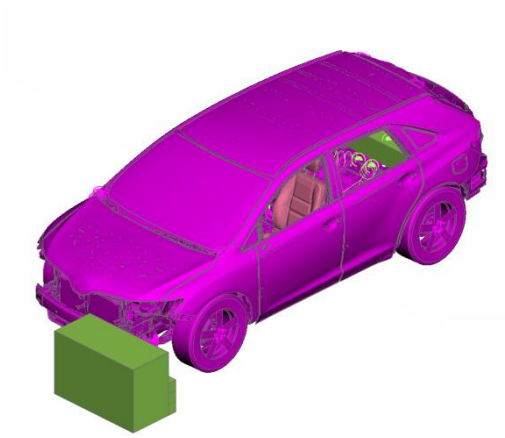
A new Draw Mode is introduced in ANSA that enables users to visually review the modularization of models.



There are two Draw Modes under the MBCONTAINER option, Subsystem and Classification.



In the Subsystem draw mode, each subsystem – library item is highlighted with a different color.



In the Classification draw mode there is the possibility to visually group the model according to the different model browser containers. For example the model can be highlighted according to the Simulation Model and the Loadcase.

3. Modular Storage

With the aid of the ANSA Data Management functionality, the modular model organization discussed in Chapter 2 is also reflected on the disk. The low level units of the modular organization are stored as monolithic files and are then referenced by different compound entities, enabling data reusability while, at the same time, optimizing the disk space required for storage.

Two data management backend options are possible in the Modular Environment:

- File-based ANSA DM: Suitable for smaller scale implementations, or as an entry-level solution, the file-based ANSA DM is ideal for small teams, to deal with the management of the model and simulation data produced daily and provide an environment for effective collaboration between the team members. ANSA DM is embedded in ANSA which means that it's already available to all ANSA users and requires no extra license or cost.
- SPDRM: Suitable for larger-scale, Enterprise level implementations, SPDRM is a server-client application that uses a data server for data storage. It is a separate program that requires a special license and has its own installation procedure. It can be utilized at team, division, or department level, and it also offers capabilities for the collaboration and data synchronization among geographically dispersed teams.

No matter which option is used as a Data Management backend, the functionality used for pushing data to the storage and downloading data from the storage in the running ANSA (or META) session is identical.

3.1. Data I/O: Main principles

3.1.1. Data Objects

Data objects in DM, usually referred to as DM Objects or DM Items, are sets of information that can fully describe a Model Browser Container and they consist of the following:

For all Model Browser Containers:

- **Primary attributes** or **Properties**: Set of metadata that has been declared as the identifying keys of a Model Browser Container type in the DM Schema. No two DM Objects in the data repository can have the same set of primary attributes. Similarly, within an ANSA session, items with the same set of primary attributes will get merged (or become instances)
- **Secondary attributes** or **Attributes**: Set of metadata that should always be available, but do not take part in the identification of the items. These attributes are also declared in the DM Schema, in order to ensure they will be available at all times. A characteristic example is the attribute for *Status*. Note that these attributes can also be of types FILE or DIRECTORY in order to associate a Model Browser Container with one or more external files or directories that are always managed as a bundle.
- **Additional attributes**: Any other metadata that are either auto-assigned by the Data Manager (e.g. DM Creation Date, File Size, Owner) or are pushed by the application that creates a DM Object. For example, all User Attributes of a Model Browser Container type that have been marked as "Save in DM=Yes" and have a non-empty value, will be automatically saved as additional attributes of the DM Object. The same applies for the ANSA characteristics of the Model Browser Containers, like mass, inertia and size info.
- **File**: This is the representation file of the Model Browser Container, e.g. the ANSA file of a Subsystem with File Type = ANSA, or the main keyword file of a Simulation Run with File Type = LS-Dyna. The File is always declared as an attribute of type ATTACHED FILE in the DM Schema. Depending on the case, this attribute may be declared as a Primary or Secondary attribute. This attribute is mandatory for all Model Container

Types, no matter if the target modular structure of the model will indeed require an independent file for all Model Container Types. For file-based ANSA DM in particular, this attribute is not required for Parts and Subsystems.
Note that since there may be more than one attribute of type ATTACHED FILE declared for a Model Container Type in the DM Schema, there's the convention that the attribute that refers to the main representation file of the Model Browser Container must always be defined last.

For compound Model Browser Containers:

- **Hierarchy**: The hierarchical structure of Model Browser Containers that consist of other Model Browser Containers. This hierarchical structure is communicated in the form of an XML file. A hierarchy may describe the composition of a subsystem of parts and groups, the composition of a simulation model of subsystems, the composition of a simulation run of a simulation model and a loadcase, etc. This information is parsed by the Data Manager in order to construct the contents-type relationships between DM Objects, allowing the preview of the contents of the compound DM Objects directly in the data browser (i.e. DM Browser or SPDRM client)
- **Definition ANSA File**: This file is saved automatically by ANSA every time a compound Model Browser Container of solver File Type is uploaded to DM. It is uploaded to DM as an additional attribute of the Model Browser Container. It contains the empty placeholders of the children Model Browser Containers that construct the saved object together with any entities that may be directly assigned to the saved object like for example connectors, assembly sets, etc.

3.1.2. Definition ANSA Files

Typically, in order to maintain important features of the ANSA model after its output in solver keyword format, ANSA comments are written at the end of the solver keyword file. The use of ANSA comments could protect entities like connections, GEBs or part hierarchy from being lost and would enable the analysts to edit an include in ANSA and still have access to all useful ANSA entities.

With the growing complexity of models, the amount of the ANSA-specific metadata that need to be maintained after output of the model has increased tremendously, leading to the block of ANSA comments, in many cases, being considerably bigger in size than the block of actual solver keywords. On top of this, there are still ANSA definitions that are impossible to capture in the form of ANSA comments (e.g. Features, or Model Browser Container structure). Due to these reasons, a standalone ANSA file is introduced in order to keep the "ANSA representation" of solver keyword files. This ANSA file goes "hand-in-hand" with the solver keyword file and all functions of the Modular Environment recognize this file as the definition file of the Model Browser Container.

Depending on the Model Browser Container, the definition file may either be bundled under the DM Object with solver File Type or may be handled as an independent DM Object that is linked to the solver File Type DM Object with a characteristic link.

In any case, the *Definition ANSA File* is the reference file for the creation of a new iteration of a DM Object. It contains all "rich" ANSA definitions that are needed by an analyst in order to work efficiently with the model. Therefore, every time the analyst starts a process for the creation of a new simulation loop through the DM Browser function *Next Iteration*, ANSA will load the **Definition** of the selected DM Object, regardless of the File Type of the DM Object selected.

The contents and composition of *Definition ANSA Files* in each case are described below:

Definition file bundled with the solver DM Object

For all compound Model Browser Containers, the definition file is stored as an attachment to the DM Object with solver File Type. This means that a Simulation Run with File Type = Nastran will have an additional attribute named *Definition ANSA File* that will hold the ANSA file that contains the run definition.

The *Definition ANSA Files* contain the structure of contained Model Browser Containers as well as any Model Setup Entities and Intermodular Connections that are assigned to the compound Model Browser Container.

During “Save in DM” of compound Model Browser Containers consisting of 2 or more levels of children, the content of the *Definition ANSA File* may be written in a modular way, depending on the composition settings specified for Save.

For example, saving a Simulation Run with *monolithic* Simulation Model and Loadcase will lead to the creation of a single *Definition ANSA File* for the Run that will contain all Model Browser Containers bottom-up. So, in order to load the complete structure of the Simulation Run in ANSA at a later time, one would have to read a single *Definition ANSA File*. However, saving a Simulation Run *with references* (or with *Reference inner contents)* to the Simulation Model and Loadcase, will lead to the creation of a *Definition ANSA File* for the Run that will only contain references to the Simulation Model and the Loadcase and no reference to their contents since it is assumed that the Simulation Model and Loadcase composition will become available through the *Definition ANSA Files* of their own DM Objects. Thus, in order to load the complete structure of the Simulation Run in ANSA at a later time, one would have to merge 3 different *Definition ANSA Files*: The one of the Simulation Run and then those of the Simulation Model and the Loadcase.

The tables below show the different possibilities for two usual cases of Model Browser Containers that contain two or more levels of children. The first case is the Simulation Run, which consists of Simulation Model and Loadcase which, in turn, contain Subsystems and Library Items. The second case is the Simulation Model, in case Subsystem Groups are also used for model organization. In these cases the Simulation Model contains Subsystem Groups which, in turn, contain Subsystems.

Simulation Run: Composition of *Definition ANSA File*

Contents option: Monolithic/Monolithic	**Contents option: with References**
Simulation Run Model Setup Entities Simulation Model Model Setup Entities Inter-modular connections Subsystems Loadcase Model Setup Entities Inter-modular connections Library Items	**Definition ANSA File of Run** Simulation Run Model Setup Entities Loadcase Simulation Model **Definition ANSA File of Loadcase** Loadcase Model Setup Entities Library Items **Definition ANSA File of Simulation Model** Simulation Model Model Setup Entities Inter-modular connections Subsystems

Simulation Model: Composition of *Definition ANSA File*	
Contents option: Monolithic/Monolithic	**Contents option: with References**
Simulation Model Model Setup Entities Inter-modular connections Subsystem Groups Model Setup Entities Inter-modular connections Subsystems	**Definition ANSA File of Simulation Model** Simulation Model Model Setup Entities Inter-modular connections Subsystem Groups **Definition ANSA File of Subsystem Group** Subsystem Group Model Setup Entities Inter-modular connections Subsystems

In case of Loadcases that involve Header Builder Library Items, it is possible to include the content of the Header in the definition file of the Loadcase. This would mean that the Header/Step setup as well as the Boundary Conditions it contains would be stored directly in the definition file of the Loadcase. This behavior is controlled through the ANSA.defaults with the setting SaveLoadcaseAssistantEntitiesAlongWithHierarchyLoadcase which is inactive by default. The *Definition ANSA File* in the two cases is shown in the table below.

Loadcase: Composition of *Definition ANSA File*	
Setting value: false *(default)*	**Setting value: true**
Loadcase (from Header Builder) Model Setup Entities Header Builder	Loadcase (from Header Builder) Model Setup Entities Header Builder Nastran Header/Abaqus Step

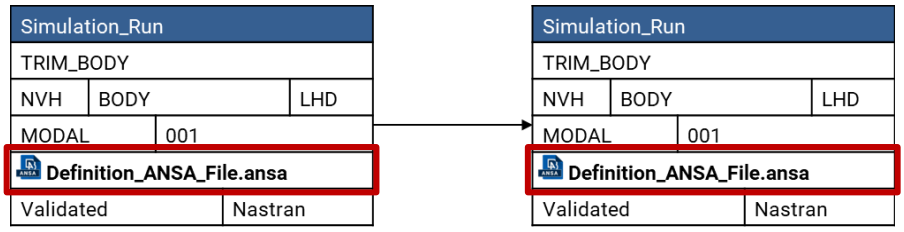
Definition file independent from the solver DM Object

For base level Model Browser Containers like Subsystems, the Modular Environment assumes that the definition file of a solver keyword file Subsystem is a Subsystem that has identical primary attributes and File Type = ANSA. This file must be saved manually by the user as a monolithic ANSA file. Note that in case a Subsystem solver file is not saved as a monolithic file but is saved with references to its contained parts, it is considered as a compound Model Browser Container and thus, its definition file is bundled with the solver DM Object.

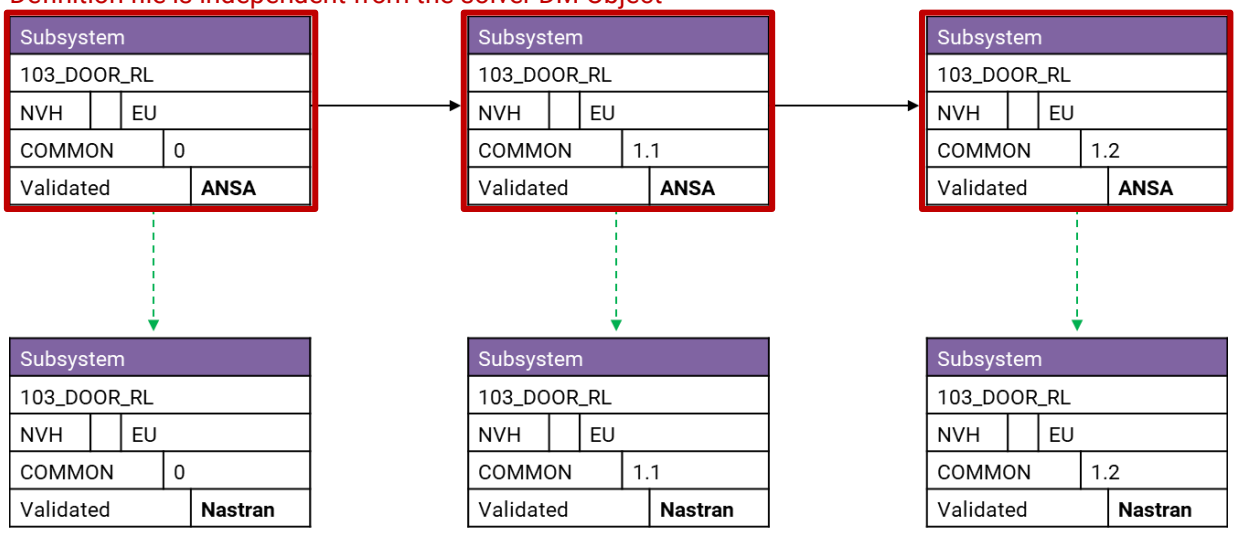
In any case, given the solver DM Object, the Modular Environment is aware of which is the Definition ANSA File.

A schematic representation of the two cases is shown below, for Simulation Runs and Subsystems.

Definition file is bundled with the DM Object in the form of attribute

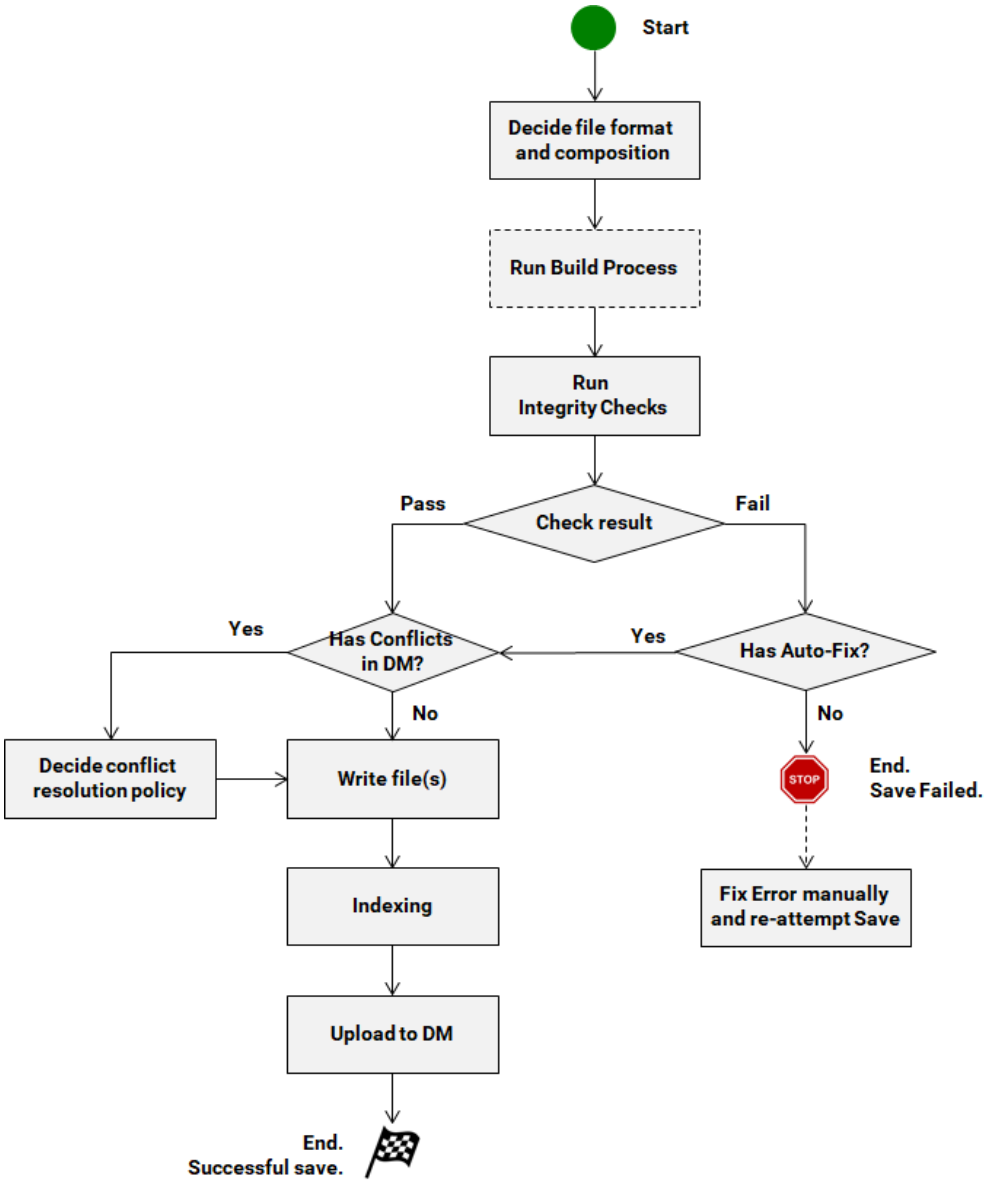


Definition file is independent from the solver DM Object



3.1.3. Pushing data to storage

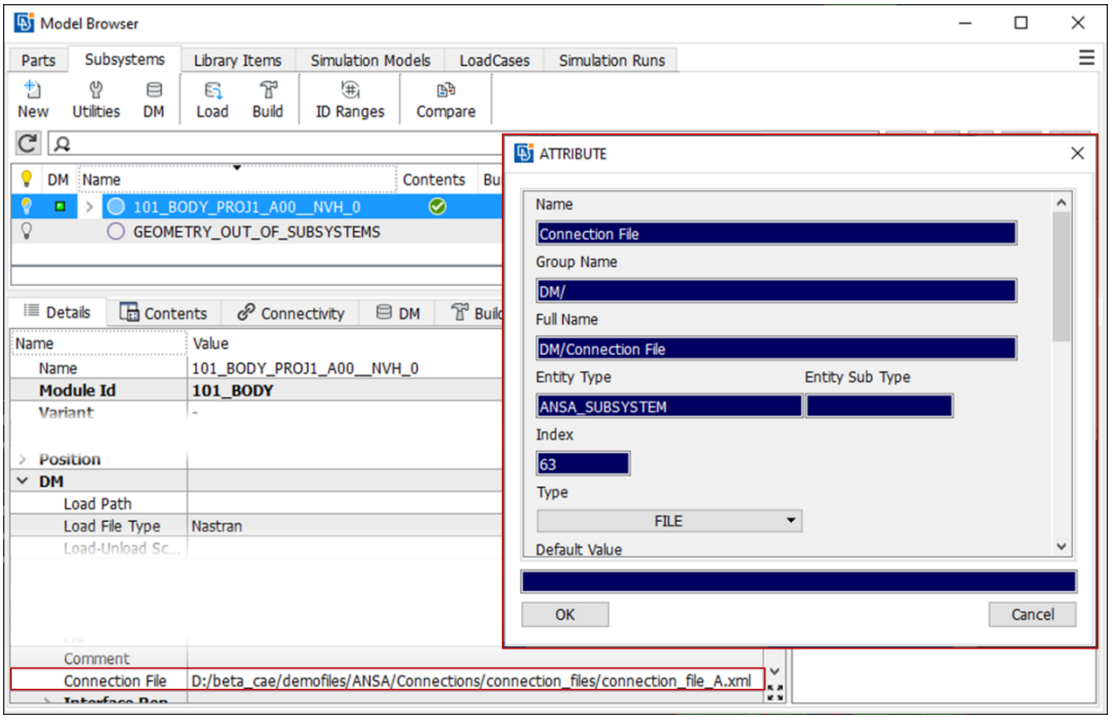
The action of pushing data from the running ANSA session to the storage is usually referred to with the term “Save in DM” and is handled either from within the Model Browser, or through the Python Scripting API. The Save action includes several steps, as shown below:



- In the beginning, the Modular Environment collects important save settings that control the format and composition of the saved item and instructions on whether to run the Build Process on Save or not (for more info, see paragraphs 3.3 and 3.7)
- Based on these settings, a set of Integrity Checks is executed, to verify the integrity of the module being saved (for more info, see paragraph 3.4)
- Then, the Data Manager communicates any conflicts between the incoming object and existing objects in the database and ANSA offers proper conflict resolution options (paragraph 3.5)
- Next, the main representation file of the Model Browser Container is generated.
- Then, the indexing process runs, which creates additional secondary attributes that are filled automatically with information related to the content of the the saved model unit (paragraph 3.2)
- Finally, the main file is uploaded to storage together with all the metadata that are collected from the Model Browser Container’s attributes.

The representation file is saved temporarily in the user's local BETA cache folder and is then uploaded to the final destination in the data repository (the final destination differs depending on the data management backend).

It is likely that during a Save operation, more than one files are uploaded to DM for a particular Model Browser Container. This can occur in case the Model Browser Container has been associated with external files or directories through attributes of type "FILE" or "DIRECTORY" as shown in the screenshot below.



3.1.4. DM Header

The DM header is a specially formatted block of comment text that is prepended at the top of the Main Representation keyword file of a Model Browser Container. It describes the metadata of the Model Browser Container, mainly Primary attributes but also some Secondary and optionally Additional Attributes. By default, the DM header includes the primary attributes. However, it is possible to add more Model Browser Container attributes to the DM header (e.g. the Status or Comment), by adding their names comma-separated, under the ANSA defaults variables:

- `SubsystemAdditionalAttributesInDmHeader`
- `SimulationModelAdditionalAttributesInDmHeader`
- `LoadCaseAdditionalAttributesInDmHeader`
- `SimulationRunAdditionalAttributesInDmHeader`

Example: DM header of a Simulation Model

```
$============================DM INFO BEGIN============================
$ANSA_DM_INFO_PRE_FORMAT;
$
$Entity Type     : Simulation Model
$Discipline      : crash
$Model Id        : pedestrian_assembly
$Model Variant   : -
$Project         : METRO
$Release         : R1
$Iteration       : 001
$File Type       : LsDyna
$File            :
$Name            : pedestrian_assembly_METRO_R1__crash_001
$User            : demo
$Solver Version : R11.1
$
$END_ANSA_DM_INFO_PRE_FORMAT;
$============================DM INFO END============================
$
```

Through the DM header, the user can understand the identity of an object when text editing its file. Furthermore, the DM Header is used by ANSA when reading a keyword file in order to regenerate the corresponding Model Browser Containers. In case a compound Model Browser Container is saved as a monolithic keyword file, several DM headers are written in the file, at the beginning of each block of keywords that corresponds to a Model Browser Container. Therefore, even in this case, it is still possible to regenerate the Model Browser Containers by reading a monolithic solver keyword file.

3.1.5. Downloading data from the storage

Data stored in the repository can be downloaded in the running ANSA session through the DM Browser, either after marking and running the action “Download” or with drag’n’drop from the DM Browser in the ANSA drawing area. Depending on the nature of the download request, the Data Manager may export different combinations of the following information:

DM Object definition: This is the description of the DM Object as recorded in the database of the Data Manager, excluding any files that were possibly attached to it. The database record consists of all primary, secondary and additional attributes with their values. This information is communicated from the Data Manager to ANSA through an xml file. Loading the definition of a DM Object in ANSA leads to the creation of an empty Model Browser Container in the Model Browser.

DM Object hierarchy: This is the description of the hierarchical structure of the DM Object. All DM Object Types have hierarchy: Starting bottom-up, the hierarchy of Subsystems and Library Items consists of the parts and groups they contain, the hierarchy of Simulation Models and Loadcases consists of Subsystems and Library Items and finally the hierarchy of Simulation Runs consists of a Simulation Model, a Loadcase and possibly some Library Items.

Interface Representation file: This file may only be available for Subsystems and Library Items. It is an ANSA file, generated automatically by ANSA during the indexing process to hold Interface information (Assembly and Loadcase Points, Interface Sets, etc.)

Definition file: This file may be available for Subsystems and all compound Model Browser Containers as described in paragraph 3.1.2.

Main Representation file: This file is available for all Model Browser Containers that are saved as ANSA files or solver keyword files, monolithic or with references.

The available options depend on the type of the DM Object to be downloaded. The list of available options for base level Model Browser Containers is given below:

	DM Object definition	DM Object hierarchy	Interface Representation File	Definition File	Main Representation File	Details
Part						
Download	✓				✓	The DM Object definition is read together with the main representation file. If there are discrepancies in the Model Browser Container metadata between the two, the former prevails.
Subsystem						
Download (File Type=ANSA)	✓				✓	The DM Object definition is read together with the main representation file. If there are discrepancies in the Model Browser Container metadata between the two, the former prevails.
Download (File Type=Solver)	✓		✓		✓	The DM Object definition is read together with the main representation file and the Interface Representation File. The latter holds all information related to the interfaces of the Subsystem and is required in cases the Subsystem interfaces were marked as “Output with Subsystem = No”. In such cases, the interface elements and the interface markers only exist in the Interface Representation File and not in the Main Representation File.
Interface Representation			✓			
Definition File				✓		For a Subsystem, this option is only available in case its Definition ANSA File is bundled with the solver File Type DM Object.
Parts Hierarchy		✓				
Object Definition	✓					
Next Iteration				✓		For a Subsystem, this option is only available in case its Definition ANSA File is bundled with the solver File Type DM Object.
New > From DM	✓					

Similarly, the list of available options for compound Model Browser Containers is given below:

	DM Object definition	DM Object hierarchy	Interface Representation File	Definition File	Main Representation File	Details
Subsystem Group / Simulation Model / Loadcase / Simulation Run						
Download		✓			✓	The DM Object hierarchy file is read together with the main representation file. The DM Object hierarchy encapsulates the information available in a DM Object definition file for base Model Containers and therefore, if there are discrepancies between the DM Object definition metadata and the Main Representation file, the former prevails. The Model Browser Container structure is rebuilt with the aid of the DM Object hierarchy.
Object Definition	✓					
Definition File				✓		
Next Iteration				✓		The definition file of the DM Object is read.
New > From DM				✓		The definition file of the DM Object is read.

	DM Object definition	DM Object hierarchy	Interface Representation File	Definition File	Main Representation File	Details
Library Item						
Download (File Type=ANSA)	✓				✓	The DM Object definition is read together with the main representation file. If there are discrepancies in the Model Browser Container metadata between the two, the former prevails.
Download (File Type=Solver)	✓		✓		✓	The DM Object definition is read together with the main representation file and the Interface Representation File. The latter holds all information related to the interfaces of the Library Item and is required in cases the Library Item interfaces were marked as “Output with Subsystem = No”. In such cases, the interface elements and the interface markers only exist in the Interface Representation File and not in the Main Representation File.
Interface Representation			✓			
Object Definition	✓					
New > From DM	✓					

3.2. Indexing

In all cases when a user adds some data in DM, either with “Save in DM”, where the file is generated on the spot by ANSA, or with “Add in DM”, where an existing file is copied to DM as a Subsystem or Library Item, ANSA scans it in order to extract useful information on its contents and add it as metadata to the generated DM object.

These metadata are stored as Additional Attributes of the DM object and can provide valuable information about each DM object directly, eliminating the need for loading the representation file in ANSA.

3.2.1. Categories of indexing metadata

There are two categories of indexing metadata, as shown in the tables below.

Category 1: Definition Attributes			
Attribute Name	**Description**	**Save in DM**	**Add in DM**
Nodes / Elements / Sets / Properties / Materials / Define / Function / Rest Min / Max Id	The id ranges of the entities of the file, categorized in the 8 types that can be used for id offsetting in LS-DYNA	✓	✓ (1)
ID Ranges Excluded Entities	The ids of nodes that are excluded from the calculation of the node id range. These are nodes marked by interface points that request particular node id that lies outside the id range of their base module.	✓	✗
Undefined Entities	The name and id of all undefined entities found in the file, except for Parameters, that are stored in a separate attribute	✓	✓
Undefined Parameters	The name of parameters that are used in the file but are not defined in the file	✓	✓
Defined Parameters	The name and value of parameters and parameter expressions that are defined in the file	✓	✓
Unsupported Numbering	A boolean indication of whether the numbering of entities in the file will lead to automatic renumbering when reading in ANSA	✗	✓
Unsupported Keywords	A boolean indication of whether the file contains unsupported keywords	✗	✓

(1) When adding a file in DM, even in the cases where the entities numbering within the file is not directly supported by ANSA (e.g. LS-DYNA files that share the same ids between elements of different types or between sets of different types), the original ids of the file will be stored as Definition Attributes and not the ids of the file that one would see in ANSA after loading it (where renumbering would inevitably take place).

Category 2: Interface Attributes			
Attribute Name	**Description**	**Save in DM**	**Add in DM**
Interface Representation > A-Points [1]	This set of attributes describes the interface points to be used for assembly.	✓	✓
Interface Representation > LC-Points[1]	This set of attributes describes the interface points to be used for loadcase setup.	✓	✓
Interface Representation > Interface Sets[1]	The Name, Id and Label of sets marked as Interface Sets	✓	✓
Interface Representation > Properties[1]	The Name and Id of properties marked as Interface Properties	✓	✓
Interface Representation > BC Sets[1] (NASTRAN only)	The type and Id of B.C. SETs	✓	✓
Interface Representation File	An ANSA file containing all interface entities of a Subsystem or Library Item. Interface Representation Files are used during Smart Assembly and Load-case set-up, in order to provide “rich” information about the Interfaces to the Assembly or Load-case set-up methodologies. They contain all types of interface entities plus B.C. SETs information.	✓	✓[2]

[1] Note that the interface representation attributes are zipped in order to reduce the size of database records, i.e. a single additional attribute packs information of 20 interface entities at a time (zipped attributes are named EncodedInterface_0001_0020, etc.)

[2] In case the file is an LS-DYNA keyword file containing unsupported numbering of sets, the creation of the Interface Representation file is skipped.

3.2.2. Usage of indexing metadata

Indexing metadata are used by the Build Actions on the compound Model Browser Containers or on Adapters. More specifically:

Category	Attribute Name	Used by
Definition	Nodes / Elements / Sets / Properties / Materials / Define / Function / Rest Min / Max Id	These attributes are used by the “ID Handling” Build Action and provide information on the source id ranges of the module, which are referred to by ANSA at all times a target id range is to be defined. For more information, see paragraph 5.2.1.
	Undefined Entities	These attributes are not used by any processes downstream.
	Undefined Parameters	These attributes are used in the “Parameters” bottom tab of the Model Browser and by the “Check Parameters” Build Action.
	Defined Parameters	
	Unsupported Numbering	If any of these two attributes is True for a Subsystem or Library Item, the moment it is loaded in ANSA, it is marked automatically as “Read-Only”.
	Unsupported Keywords	
Interface	Interface Representation File	The file associated with this attribute can be loaded through DM>Load Interface Representation and can be used for Smart Assembly and Load-case set-up.

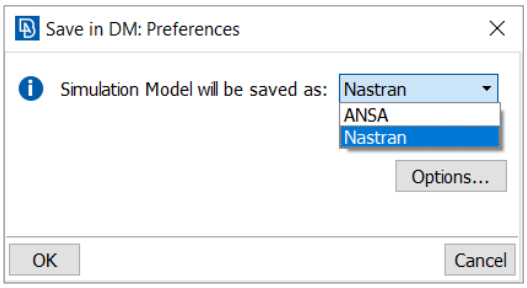
3.2.3. Indexing metadata on Model Browser Containers

The indexing metadata of a base module are kept under the *Definition* group of attributes in the Model Browser. These attributes are filled:

1. When a base module is saved in DM
2. When the representation file of a base module is loaded in ANSA from DM
3. When the object definition of a base module is loaded in ANSA from DM (empty placeholder)
4. Just in time, when required for the set-up of adaptation, in case the module is loaded in ANSA

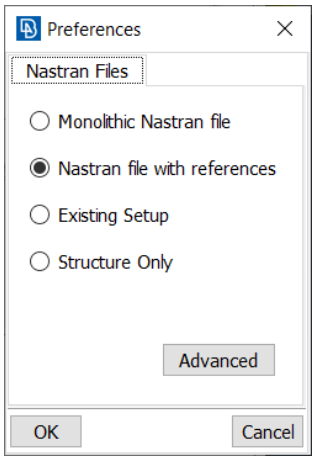
These attributes won’t be updated real-time after model modifications. However, they will be updated automatically the moment an “adaptation” tool will attempt to retrieve their values, as described in paragraph 5.3

3.3. Important Save Settings



*All types** of Model Browser containers can be stored as ANSA and solver keyword files. "Save in DM" process pops-up a dialog for the definition of the desired file type as a first step of the save process.

**Library Items can be saved only as solver files, with the exception of Loadcase Headers and Target Points which can be saved only as ANSA files. Also, Loadcases with Target Points can only be saved as ANSA files.*



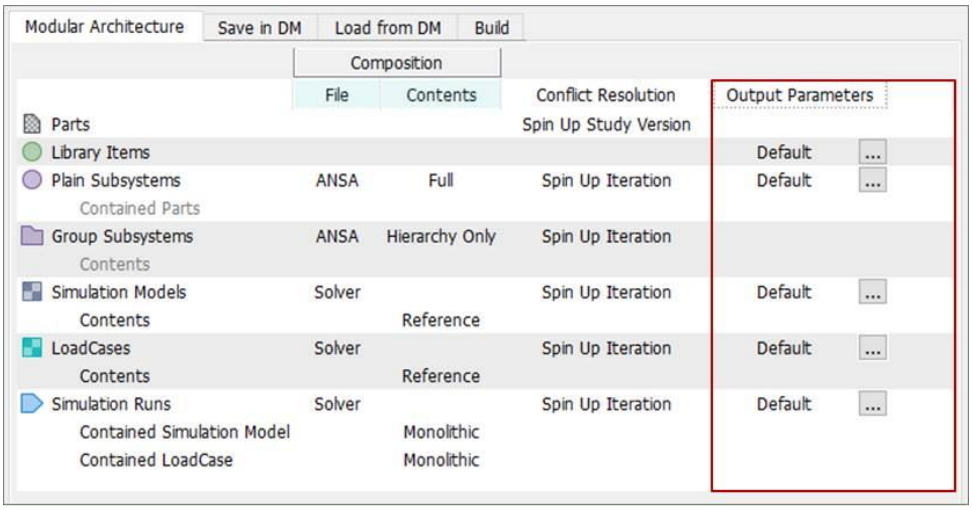
By pressing the **Options...** button, the user is able to control the composition of the file. For compound Model Browser containers being saved as solver keyword files, it is possible to select among *monolithic file*, containing the contents of its child Model Browser Containers directly in its file, or alternatively as a *file with references*, which corresponds to a file with include keywords to the include files of all child Model Browser Containers.

The option *Existing Setup* can be used when the user would like to get a keyword file where some of the base Model Containers will be referenced as include keywords and other will be written inline. In this case, as described in paragraph *5.2.4 "Adapting attributes for the control of the structure of the main files"*, the user should define accordingly the **Output Option** attribute of each Simulation Model, Loadcase or Simulation Run Adapter.

Access to the Deck Output Parameters window is given through the **Advanced** button in order to view or change the output options.

The default settings for these dialogs can be set in the *Modular Architecture* tab under *ANSA Settings > Modular Environment*. In the *Composition* section, the options for the *File* and the *Contents* can be defined. Note that the equivalent of *Existing Setup* option above is the *Respect "Output Option"* in the *Contents* column.

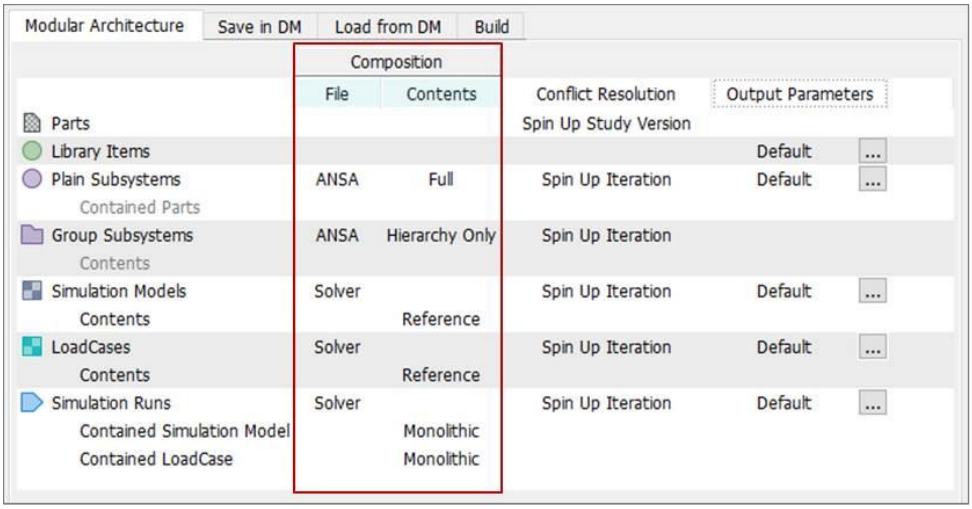
The Deck Output Parameters for each different type of Model Browser container can be controlled independently by pressing the button ...

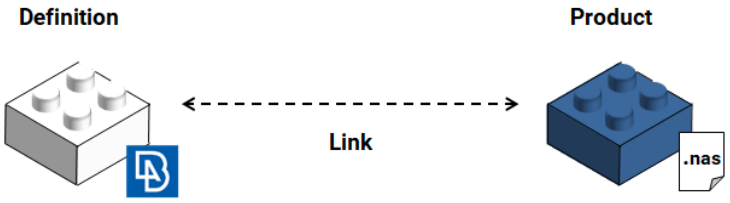


In the *Output Parameters* window, in the *Model Browser Containers* section, the user can define the desired **File extensions** for the selected Model Browser Container type.

This definition is controlled separately for each Model Browser Container, enabling the use of a different extension for base Modules (e.g. a **.k* extension for Subsystems and Library Items) and compound containers (e.g. a **.key* extension for the Simulation Model or the main Simulation Run file).

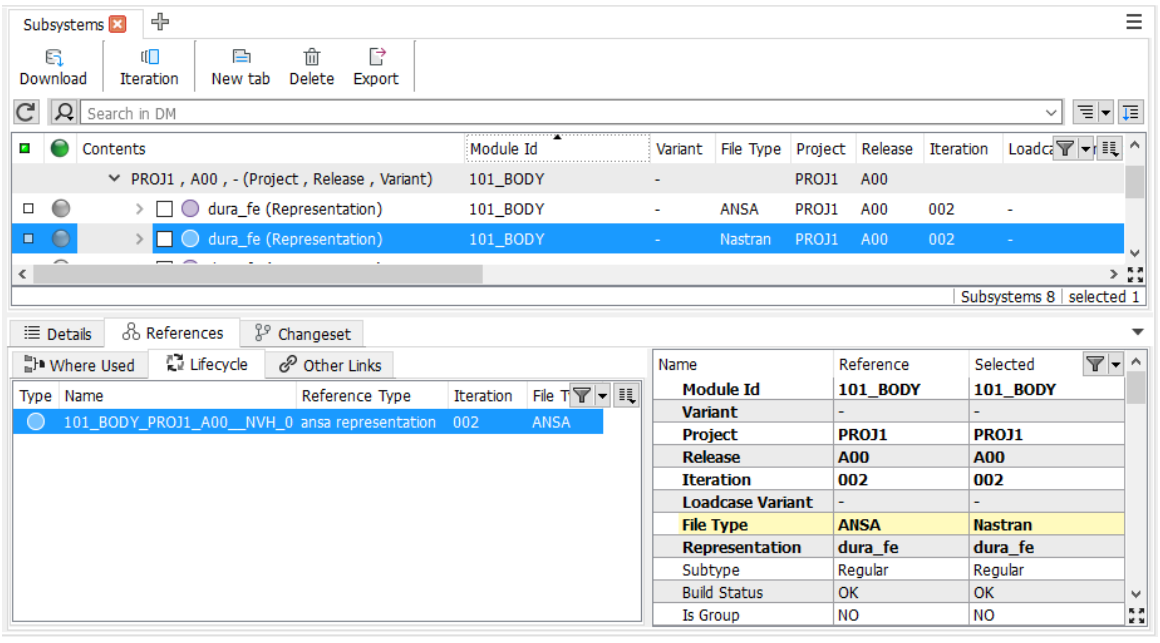
The recommended practice when working in the Modular Environment is to work mainly with ANSA files, and only generate solver keyword files “just in time” for their use in the Simulation Run. Therefore, a recommended set-up is shown in the screenshot below:



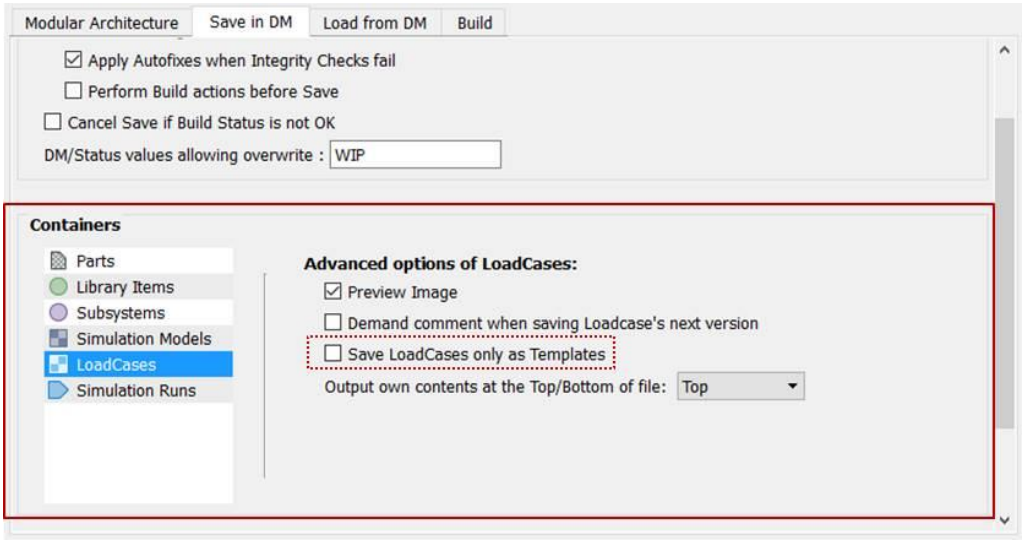


In a set-up like this, and assuming that the ANSA file of a Subsystem already exists in DM, the moment the user will attempt to save a solver keyword file with identical primary attributes, ANSA will automatically generate a link between the 2 objects in DM, tracking the relationship between the ANSA and Solver file as a Lifecycle Reference.

Such relationships can be traced in the DM Browser through the References tab. For example, selecting a Subsystem with a solver keyword File Type in the DM Browser contents, and switching to the References tab, in the Lifecycle sub-tab the linked ANSA file is listed, marked with the Reference Type “ansa representation”.



Additional settings that affect the saving of the various Model Browser Containers in DM are controlled in the *Save in DM* tab under *ANSA Settings > Modular Environment*. In the *Containers* section, the desired Model Browser Container is selected and its *Advanced options* can be set. In particular for Loadcases, it is possible to save them in DM either as Loadcase Templates that can be reused in various Simulation Runs, or as adapted entities linked with a specific Simulation Model (see paragraph *7.4. "Saving Loadcases in DM"* for more details on how to save each). Activating the option **Save Loadcases only as Templates**, the saving of Loadcase Templates can be forced also when saving adapted Loadcases.



Finally, the following two options are given during the Save of a Simulation Run:
Use loaded LoadCase: the contents of the currently loaded Loadcase will be written out as new files, or in-line, instead of using the existing in DM.
Bundle Output: all files that are referenced in the main file with include keywords will be exported in the destination folder and its references in the main file will be written in relative paths.

3.4. Integrity Checks

Name of check	Description	Auto-fix	Action to be performed
Library Items			
Properties of Library Items	Checks if all primary attributes are filled	✗	Fill all primary attributes with valid values
Subsystems - Subsystem Group			
Base checks (applied always)			
Properties of Subsystems	Checks if all primary attributes are filled	✗	Fill all primary attributes with valid values
Build Status of Container	Checks Build Status of Subsystem.(1)	✗	Apply Build action so as Subsystem gets Build Status=OK
Additional checks when Saved *with references*			
Properties of inner Parts/Subsystems	All primary attributes of the child Parts/Subsystems should be filled	✗	Fill all primary attributes of child Parts/Subsystems with valid values
All inner Parts have repr files in DM	All the contained Parts must have been already saved in DM	✗	Save all contained Parts in DM
All child Subsystems have repr files in DM	All the contained Subsystems (in case of Group Subsystem) must have been already saved in DM	✓(2)	Contained Subsystems are saved in DM
Additional checks when Saved *as monolithic*			
Subsystems are populated	Checks whether all Subsystems to be saved have assigned includes	✗	Make sure an include is associated with the Subsystem
Ids agree with the ranges of the Numbering rules	Checks whether entities ids lie inside the assigned numbering range(3)	✓	Entities are renumbered accordingly
Checks applied in special cases			
Reduced Representations status	The results file (.op2/.pch) for all the reduced representations are available (Reduced Model>Status=Ready)	✗	Run solver to produce the necessary reduced representation result file
Subsystems are not loaded as Interface Representation	Checks whether the Subsystem is loaded with its interface representation	✗	Load main representation of Subsystem.

(1) Performed if ANSA.defaults setting *cancel_save_if_build_status_is_not_ok* is active.
(2) Performed if ANSA.defaults setting *apply_autofixes_on_failures_of_integrity_checks* is active.
(3) Performed only when a Numbering Rule is defined for the selected Subsystem.

Name of check	Description	Auto-fix	Action to be performed
Simulation Model			
Base checks (applied always)			
Properties of Simulation Model	Checks if all primary attributes are filled	✗	Fill all primary attributes with valid values
Build Status of Container	Checks Build Status of Simulation Model.(1)	✗	Apply Build action so as Simulation Model acquires Build Status=OK
Additional checks when Saved *with references*			
Properties of inner Subsystems	All primary attributes of the child Subsystems should be filled	✗	Fill all primary attributes of child Subsystems with valid values
Sufficient data to create Simulation Model include file(4)	All the contained Subsystems must have been already saved in DM	✓(4)	Contained Subsystems are saved in DM
Assigned includes of Subsystems	Checks whether all inner Subsystems have assigned includes	✓	Subsystems are assigned with proper includes

(1) This check will be performed only if ANSA.defaults setting *cancel_save_if_build_status_is_not_ok* is active.

(4) This check will fail in the following cases:

- Output failure
- No Library Item/Subsystem/Simulation Model/ Loadcase file found in DM
- Error in writing unloaded content in Subsystem file
- Found include with no reference file
- Simulation Run has no Simulation Model/Loadcase
- Library Item/Simulation Model/Loadcase has not output option set
- Defined output option for Simulation Model/Loadcase is not supported
- Library Item is not loaded
- Loadcase is empty
- Library Item/Subsystem is not loaded and does not exist in DM
- Assigned include is empty

Only in case of missing files from DM, ANSA will attempt to auto-fix the problem

Loadcase			
Base checks (applied always)			
Properties of Loadcase	Checks if all primary attributes are filled	✗	Fill all primary attributes with valid values
Build Status of Container	Checks Build Status of Simulation Model.(1)	✗	Apply Build action so as Simulation Model acquires Build Status=OK
Additional checks when Saved *with references*			
Properties of inner Library Items/Subsystems	All primary attributes of the child Library Item/Subsystem should be filled	✗	Fill all primary attributes of child Model Containers with valid values
Sufficient data to create Loadcase include file(4)	All the contained Model Containers must have been already saved in DM	✓(4)	Contained Model Containers are saved in DM

(1) This check will be performed only if ANSA.defaults setting *cancel_save_if_build_status_is_not_ok* is active.
(4) This check will fail in the following cases:

- Output failure
- No Library Item/Subsystem/Simulation Model/ Loadcase file found in DM
- Error in writing unloaded content in Subsystem file
- Found include with no reference file
- Simulation Run has no Simulation Model/Loadcase
- Library Item/Simulation Model/Loadcase has not output option set
- Defined output option for Simulation Model/Loadcase is not supported
- Library Item is not loaded
- Loadcase is empty
- Library Item/Subsystem is not loaded and does not exist in DM
- Assigned include is empty

Only in case of missing files from DM, ANSA will attempt to auto-fix the problem

Name of check	Description	Auto-fix	Action to be performed
Simulation Run			
Base checks (applied always)			
Properties of Simulation Runs	Checks if all primary attributes are filled	✗	Fill all primary attributes with valid values
Properties of inner Simulation Models	All primary attributes of the child Simulation Model should be filled	✗	Fill all primary attributes of the child Simulation Model with valid values
Properties of inner Loadcases	All primary attributes of the child Loadcase should be filled	✗	Fill all primary attributes of the child Loadcase with valid values
Properties of inner Subsystems	All primary attributes of the child Subsystems should be filled	✗	Fill all primary attributes of the child Subsystems with valid values
Build Status of Container	Checks Build Status of Simulation Model.(1)	✗	Apply Build action so as Simulation Model acquires Build Status=OK
Sufficient data to create main file(4)	All the contained Model Containers must have been already saved in DM. If *references* were requested for the Simulation Model and the Simulation Model is not found in DM, its contained Model Containers are searched. If *references* were requested for the Loadcase and the Loadcase is not found in DM, its contained Model Containers are searched.	✓(4)	Contained Model Containers are saved in DM
Assigned includes of Subsystems	Checks whether all inner Subsystems have assigned includes	✓	Subsystems are assigned with proper includes

(1) This check will be performed only if ANSA.defaults setting *cancel_save_if_build_status_is_not_ok* is active.
(4) This check will fail in the following cases:

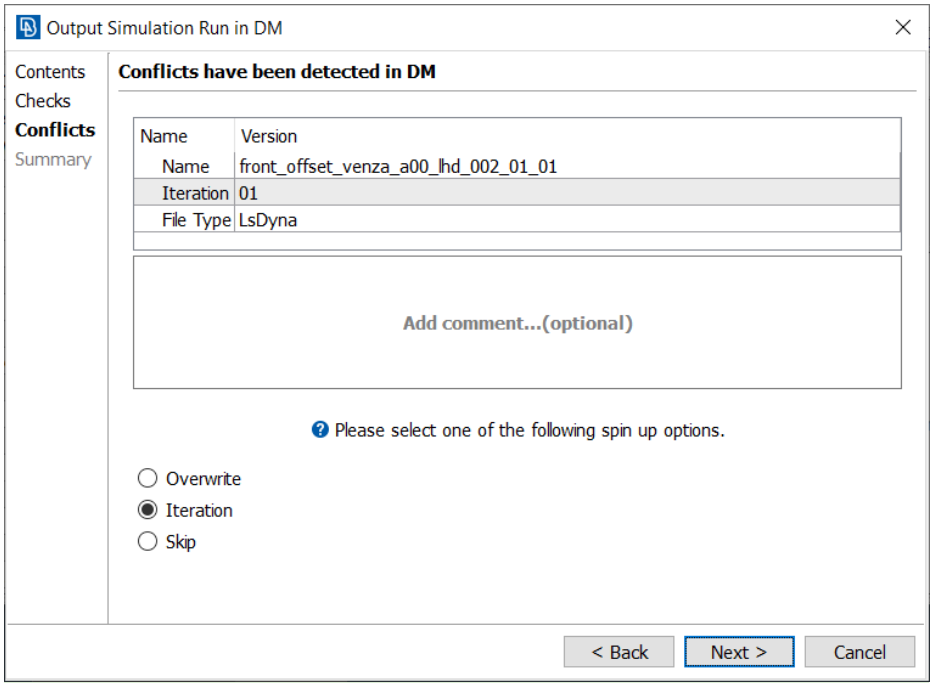
- Output failure
- No Library Item/Subsystem/Simulation Model/ Loadcase file found in DM
- Error in writing unloaded content in Subsystem file
- Found include with no reference file
- Simulation Run has no Simulation Model/Loadcase
- Library Item/Simulation Model/Loadcase has not output option set
- Defined output option for Simulation Model/Loadcase is not supported
- Library Item is not loaded
- Loadcase is empty
- Library Item/Subsystem is not loaded and does not exist in DM
- Assigned include is empty

Only in case of missing files from DM, ANSA will attempt to auto-fix the problem

Name of check	Description	Auto-fix	Action to be performed
Simulation Run			
Additional checks for Loadcases of Subtype *with Loadcase Header* in Nastran			
Loadcase compatible with selected simulation model	Checks the compatibility of the Subsystems of the model with the type of *Loadcase_Header* Library Item	✗	For Solver = Nastran: If the Subsystem is a superelement, ANSA checks the Subtype not to be *FRF Model* or *Modal Model* For Solver = META FRF Assembly: If the Subsystem is a superelement, it must be of type *FRF Model*, *Modal Model*, or *Rigid Body*
Subsystem contents required by Loadcase are available	Checks whether the contents required by Loadcase exist in the Simulation Model	✗	Load Main or Interface Representation for the Subsystems containing the affected interfaces
Checks applied in special cases			
Reduced Representations are ready for use	Checks for possible differences between database contents and files produced by solver	✓	Database contents are updated for Reduced Representations
Frf Reduced Models are combined with Display Models under the Simulation Model	Only if Simulation Model contains Reduced Model Subsystem of Reduced Model>Type=FRF MODEL, checks whether this is combined with a Display Model	✓	A minimum Display file is created, containing only interface nodes definition

3.5. Conflict detection and resolution

Within the “Save in DM” process and right after the successful execution of *integrity checks*, ANSA checks with the data manager for potential conflicts of the Model Browser Container being saved with existing DM Objects. Essentially, ANSA checks whether there is a DM Object with the same primary attributes already in DM. In case a conflict is identified, it is reported in the *Conflicts* page of the Save wizard and prompts the user to take action.



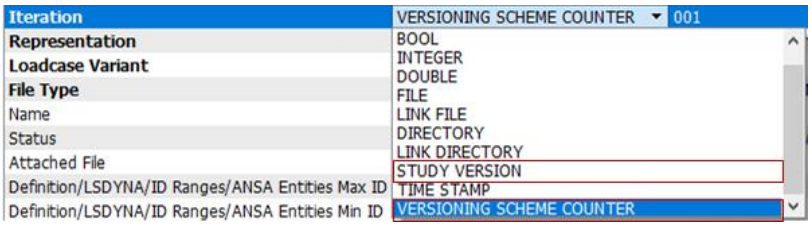
To resolve the conflict the user can:

- Save the Model Browser Container as a new version or
- Overwrite the existing DM Object or
- Halt the process

More information on each option is given in the paragraphs below.

3.5.1. Saving a new version

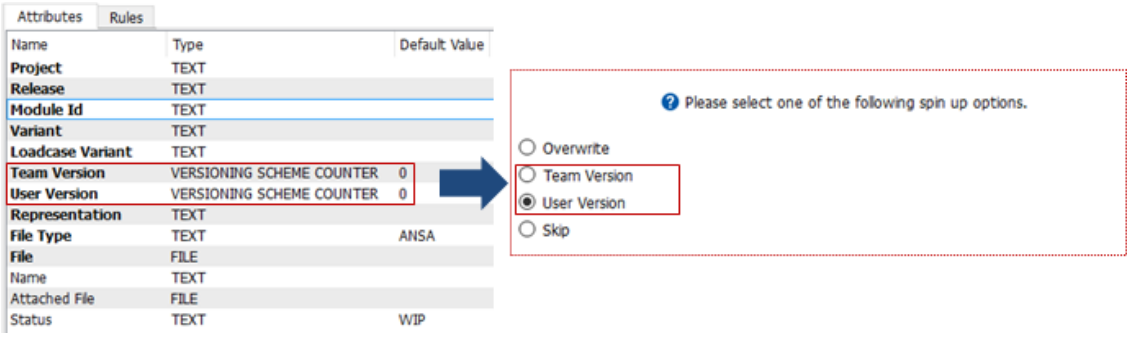
The available options for version spinup are controlled from the DM Schema. In the default DM Schema, there’s a single version attribute named *Iteration*.



Version attributes are those that have type VERSIONING SCHEME COUNTER or STUDY VERSION in the *DM Schema Editor*.

In case a custom *DM Schema* is created, that has more than one version attributes defined for an object type, the wizard UI will be adapted automatically, showing all different version attributes in the order defined.

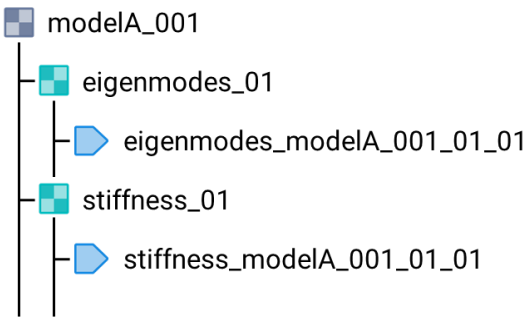
In the example below, two different versions are defined for the Subsystem: “Team Version” and “User Version”.



In a case like this, the data manager assumes a default *nesting* between the different versions defined so that when the version attribute defined first (here the “Team Version”) is incremented, all subsequent version attributes (here the “User Version”) are initialized. An example of this relationship is shown in the table below:

	Initial Versions	**Step 1** Save Subsystem Conflict resolution: Spin-up User Version	**Step 2** Save Subsystem Conflict resolution: Spin-up User Version	**Step 3** Save Subsystem Conflict resolution: Spin-up Team Version
Team Version	0	0	0	1
User Version	0	1	2	0

A similar kind of *nesting* exists in the versioning of simulation data but this time between different object types.



A Simulation Run uses a Simulation Model and a Loadcase, but a Simulation Model and Loadcase can be used by more than one Simulation Runs. This relationship is depicted in the image on the left, where the usual tree view of the Simulation Runs has been inverted. Attempting to create a new iteration of one of the Simulation Runs may lead to the creation of a new Simulation Model version and/or the creation of a new Loadcase version and/or the creation of a new Simulation Run version depending on what exactly is modified:

- In case the modifications are made in the contents of the Simulation Model, they are captured as a new version of the Simulation Model
- In case the modifications are made in the contents of the Loadcase, they are captured as a new version of the Loadcase
- In case the modifications are made in the contents of the Simulation Run (i.e. in any Library Items or Model Setup Entities assigned directly to the Simulation Run and not in the Simulation Model or Loadcase), they are captured as a new version of the Simulation Run

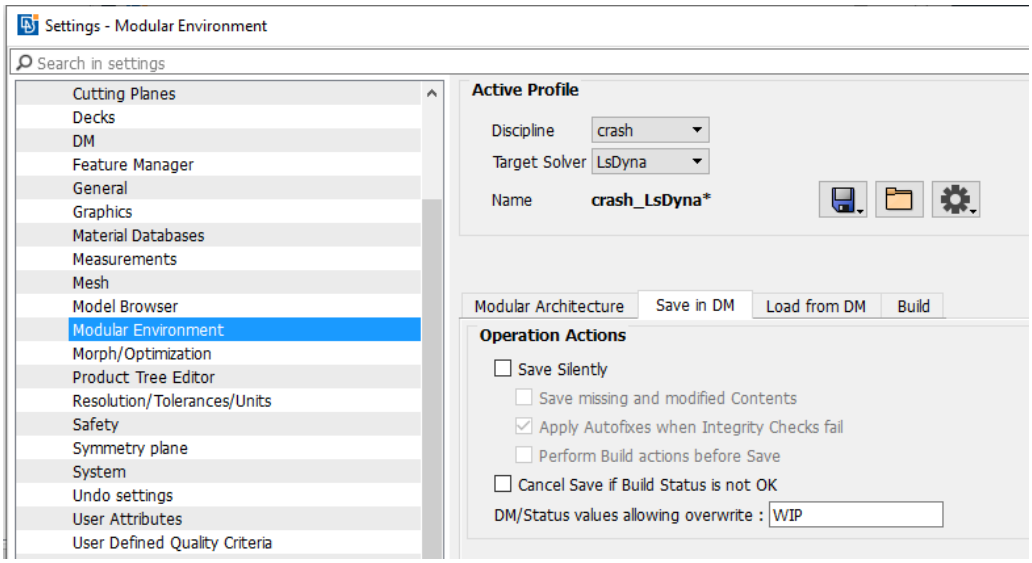
So during the saving of a run iteration, the moment a new Simulation Model version is created, the Simulation Run version attributes are initialized. Similarly, the Simulation Run version attributes are also initialized the moment a new Loadcase version is created. An example of this relationship is shown in the table below:

	Initial Versions	Step 1 Save run iteration after modification in the Sim.Model contents	Step 2 Save run iteration after modification in the Loadcase contents	Step 3 Save run iteration after modification in the Run contents	Step 4 Save run iteration after modification in the Sim.Model contents
Simulation Model Version	001	**002**	002	002	**003**
Loadcase Model Version	01	01	**02**	02	02
Simulation Run Version	1	1	1	**2**	**1**

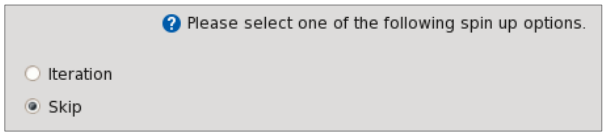
In Step 4 above, the new Simulation Model version that is created acts as a new “branch” for the overall versioning of the run and therefore, the Simulation Run version is initialized to 1.

3.5.2. Overwriting existing DM Objects

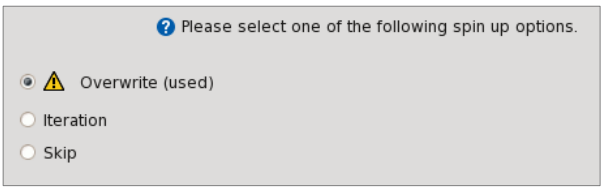
In case of file-based ANSA DM, overwrite may be restricted in case the conflicting DM Object is already used by a compound DM Object (e.g. overwrite of a Subsystem may be restricted in case the Subsystem is used by a Simulation Model DM Object). It is possible to control whether this restriction will be applied in all cases or only when the compound DM Object is of a particular maturity level and above. This is controlled through the setting *DM/Status values allowing overwrite*, found in *ANSA Settings >Modular Environment [Save in DM].*



The field accepts comma-separated values. So, in a setup like this, if the DM Object of the Simulation Model had Status OK, overwrite of the Subsystem would be forbidden and, therefore, the option would not be given in the wizard. However, if the Status was WIP, overwrite would be allowed upon user confirmation.



Subsystem being saved is used by a Sim.Model with Status = OK



Subsystem being saved is used by a Sim.Model with Status = WIP

In case of SPDRM or other 3rd party data manager, overwrite policy is controlled by the backend. Thus, the Overwrite option may appear or not, depending on the configuration of SPDRM or the data manager in general.

3.6. DM Update Status

All Model Browser Containers get a status with respect to their status in DM. This status is shown as an indication in the **DM Update Status** column in the Model Browser.

This indication is used in order for a user to:

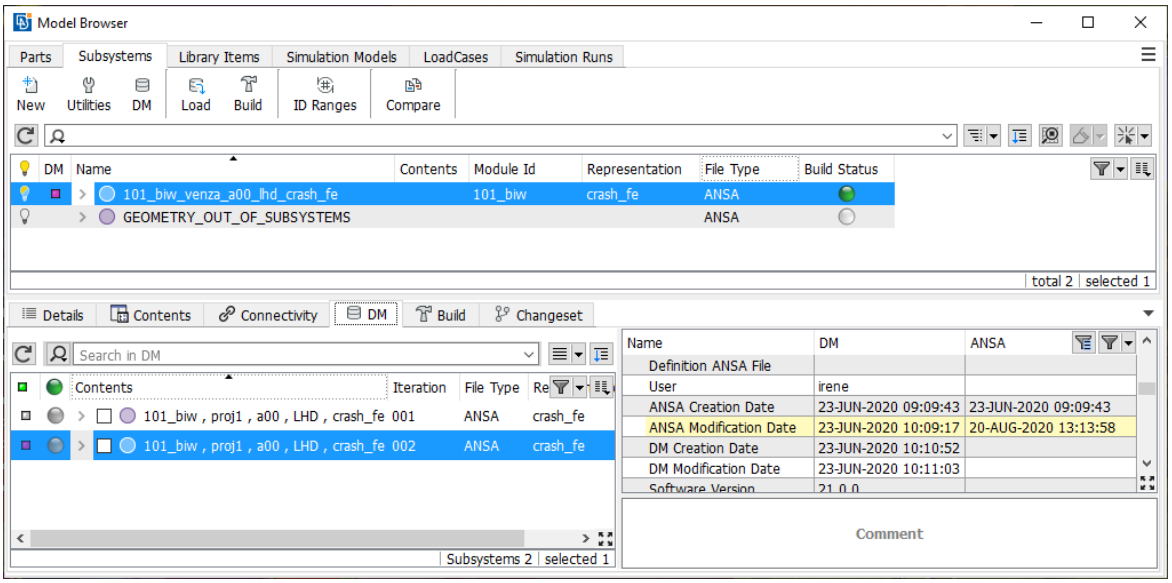
- Identify items that do not exist in DM and need to be created
- Identify items that have an update in DM and can be updated on demand
- Identify items that have been modified since the moment they were loaded in the Model Browser

All this information is communicated through 5 alternative status values, as shown in the table below:

Status	Color	Description
Up to date		The Model Browser Container is up to date
Not up to date		The Model Browser Container needs update. There is an Object in DM that either has a newer version or a newer timestamp.
Modified		This indication only concerns Model Browser Containers that exist in DM and it signifies that the item was modified since it was loaded from DM.
Alternative		Alternative representations, variants, etc. exist in DM for the entity used in the model.
Not found		The entity used in the model does not exist in DM.

These status values are very important because they determine the behavior of automatic operations like “silent save” that is described in paragraph 3.7.

Especially the “modified” indication is a means to understand which Model Browser Containers have been modified after user actions. A Model Browser Container is marked as “modified” when its “Last Edit” attribute in the Model Browser holds a timestamp newer than the “ANSA Modification Date” attribute of the corresponding DM Object, or when its parameters are modified.

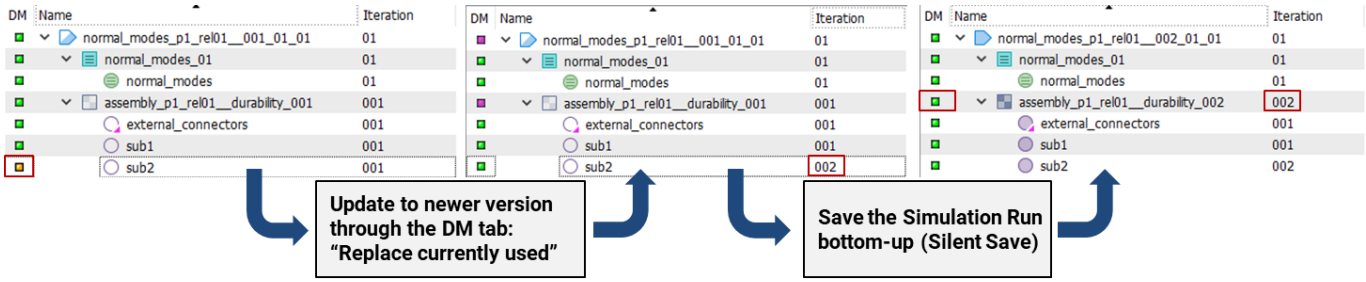


The “Last Edit” is updated every time the contents of the Model Browser Container change. Depending on the Model Browser Container, a change of the contents may be triggered by an addition, removal or modification of its contents. More specifically:

- **Change in the contents**: Add an entity in the container or remove an entity from a container. E.g. add a Face/Point/Curve/Element to a Part, add a Part/Set/Property/Contact to a Subsystem, add a Subsystem/Library Item to a Simulation Model or Loadcase, etc.
  For compound Model Containers, a change in the primary attributes of one of their contents is considered as a change in their contents. For example, changing the Iteration of a Subsystem under a Simulation Model triggers a “contents changed” event for the Simulation Model.
- **Topological changes**: Any topological change that relates to modification of Faces, etc.
- **Mesh related changes**: Any mesh related change triggered from mesh generators, mesh improvement functions, etc.
- **Edit card changes of contained entities**: Any Edit card field change emits a notification event that marks the Model Container of the entity as “modified”. E.g. a Subsystem will be marked as modified even if one of its Sets gets marked as “Frozen Delete”.
- **Thickness changes**: Even when assigned to the nodes through the element card (nodal thickness).
- **Model Browser Parameters changes**
- **Position changes**: Triggered through transformations (Transform>Move, Move Node)

For parts and Subsystems, the reasons that led to their marking as “modified” are accumulated and reported under a special entity called *Changeset*. More information about changesets is given in paragraph 3.9.

As stated above, for compound Model Browser Containers, the modification of the attributes or the adaptation of their child containers is equivalent to a change of contents. In the example below, the Subsystem is marked as “orange”, indicating the existence of an update in DM, while the Simulation Model is marked as “green”, meaning that no changes have taken place in the Simulation Model since loading it from DM. Selecting a newer Iteration for the Subsystem, the Simulation Model is automatically marked as “modified” since its contents have changed. Note that this contents change event is propagated upwards, to all higher level Model Browser Containers. This means that any Simulation Runs that make use of this Simulation Model would also be marked as “modified”.



The *DM Update Status* has two values that can be valid for the same Model Browser Container at the same time: These are the "Not up to date" (orange) and the "Modified" (magenta), since an item may have an update in DM, and, at the same time, may also be modified since loaded in ANSA. In such cases, the item is marked as "Modified" (magenta), since it is assumed that the indication of change of contents is of higher importance.

Note that the change of the contents of the Model Browser Container, except for the **DM Update Status**, also triggers the update of **Contents Status** and **Build Status** attributes. The fact that the contents of a container have changed is a good indication that the Build Process must be executed anew.

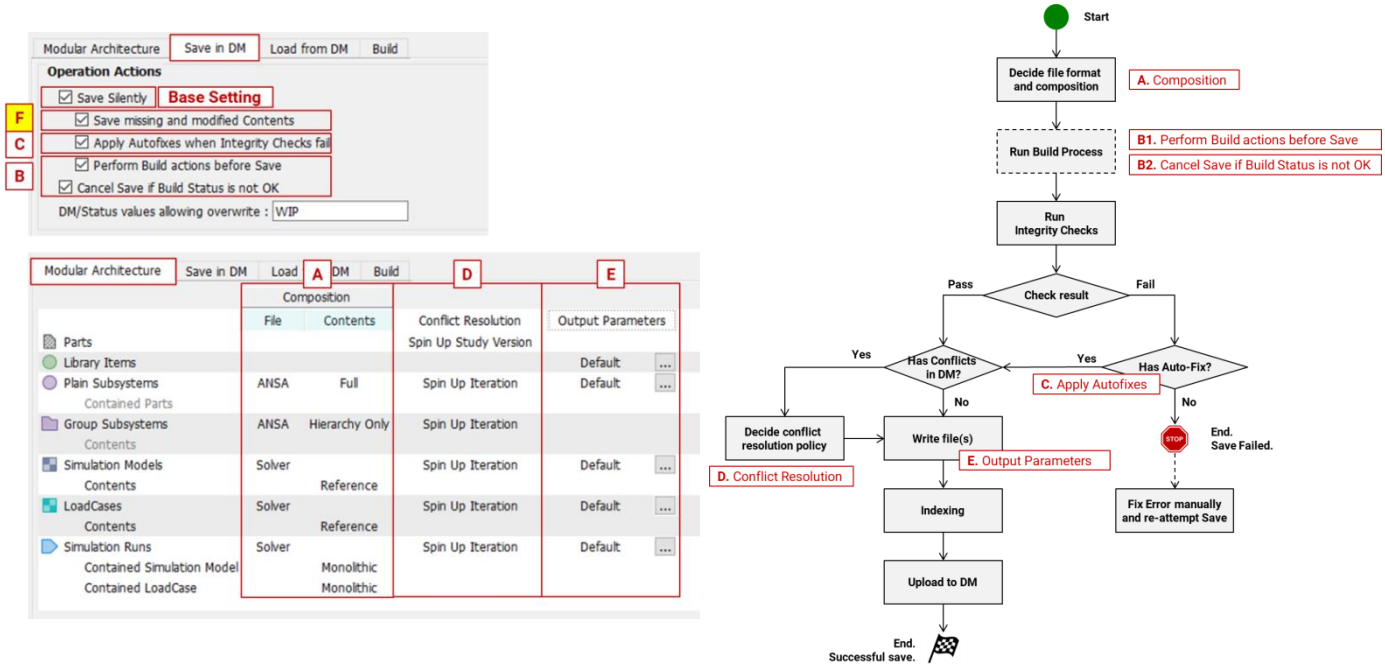
Start: Everything is up-to-date

A modification of the subsystem contents has affected all three status attributes

3.7. "Silent save"

As discussed in paragraph 3.1.3, saving data in DM is a process that involves several steps and goes through certain check points. The user needs to take decisions regarding the format and composition of the file, deal with integrity check failures, decide conflict resolution policy, etc. when working interactively, the Modular Environment offers a wizard-like "Save in DM" dialog that executes the save workflow in a stepwise manner. However, going through these options each and every time is very tedious and error prone, especially taking into account the high impact of some of these settings to the process down the road.

For this reason, the Modular Environment offers a "Silent Save" option. This is a mode which, when active, instructs ANSA to follow a pre-configured "save scenario", where all decisions are taken automatically based on a configuration sheet. "Silent Save" can be activated through *Settings >Modular Environment [Save in DM]*.



The diagram of the "Save in DM" workflow introduced in paragraph 3.1.3, shows which decisions and settings can be pre-configured through "Silent Save". The various options are explained below:

- **Save Silently**: Activates "Silent Save"
- **Save missing and modified Contents**: The "Silent Save" option enables the automatic modular save: In case the contents of Compound Model Browser containers do not exist in DM (*DM Update Status* Not found ☐) or have been modified since load (*DM Update Status* Modified ■), it is possible to trigger their saving in DM through the saving process of their parent container.
  Note that as lowest-level Model Browser Containers are considered the Subsystem and the Library Item, thus the automatic save will be performed for MBCs up to this level.
- **Apply Autofixes when Integrity Checks fail**: ANSA will execute any available autofixes in case integrity checks fail during silent save. For example, if a module only exists as an ANSA file and the user tries to save a Simulation Model in Nastran format, the integrity check "Sufficient data to create Simulation Model include file" will fail. However, having this check-box active, ANSA will auto-fix the problem by automatically creating a Nastran file from the existing ANSA file. Thus, the Save process of the Simulation Model will continue uninterrupted.

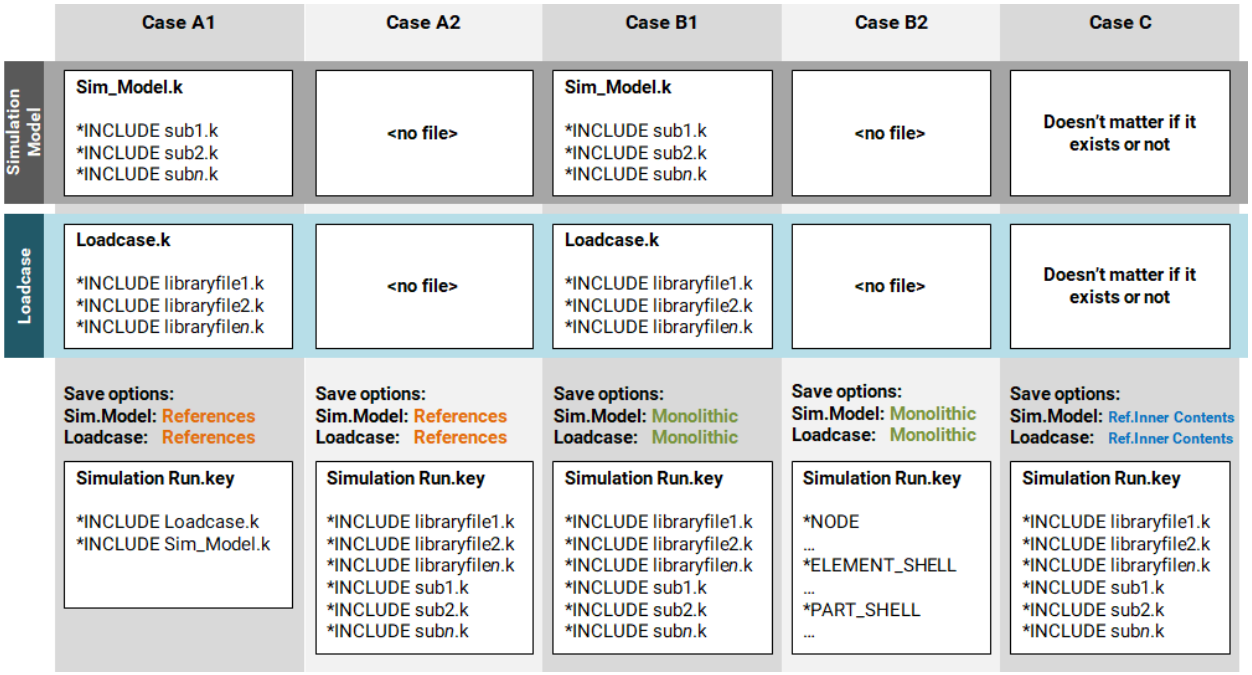
- **Perform Build actions before Save**: ANSA will execute the Build Process of the selected Model Browser Container just before Save
- **Cancel Save if Build Status is not OK**: the silent save will not proceed in case the Build Status of the Model Browser Container is not OK.

The table at the bottom can be used to define settings per Model Browser container type:

- the preferred target file type, in the column **File**
- the preferred target file content, in the column **Contents**
- the preferred target versioning policy, in the **Conflict Resolution** column
- the Output Parameters used when a Model Browser container is saved in DM as a solver keyword file, in the **Output Parameters** column. By default, output parameters only differ in the status of the "Output ENDDATA" keyword.

Additionally in the Advanced tab the possible auxiliary files that should be saved as attachments to the DM item (e.g. images, light representation files, etc.), are defined.

The image below offers a schematic representation that explains the different options available for the composition of the Simulation Run solver keyword file.



The Simulation Run contains 2 levels of nesting, i.e. it directly contains the Simulation Model and the Loadcase which, in turn, contain the Subsystems and Library Items. In this case, the interpretation of the options "References" and "Monolithic" depends on whether the Simulation Model and Loadcase are requested to be saved in DM as solver keyword files. Cases A1, A2 and B1, B2 explain the idea:

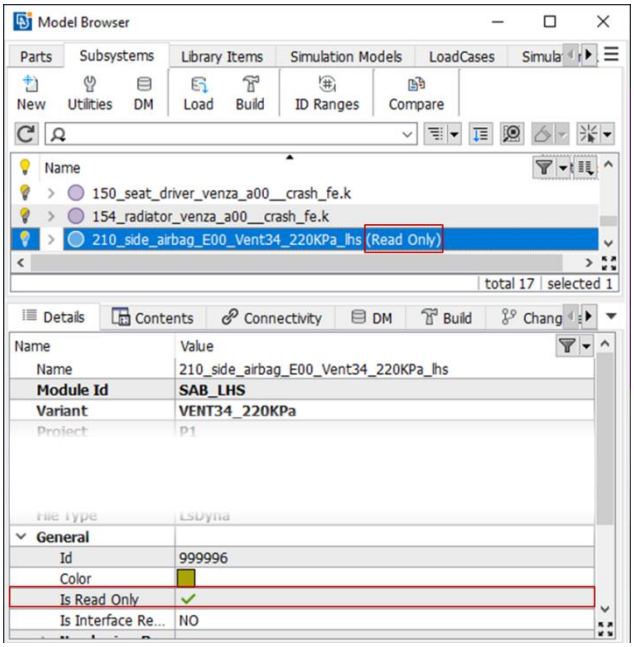
- Option "References" is requested for the Simulation Model and the Loadcase.
  In this case, the result depends on whether the Simulation Model and the Loadcase are requested to be saved as solver keyword files:
  - **Case A1**: Simulation Model and Loadcase are saved as Solver keyword files
    The Simulation Run contains only 2 include keywords
  - **Case A2**: Simulation Model and Loadcase are not saved as Solver keyword files
    The Simulation Run contains references to the contents of the Simulation Model and the Loadcase.

- Option "Monolithic" is requested for the Simulation Model and the Loadcase.
  In this case, the result depends again on whether the Simulation Model and the Loadcase are requested to be saved as solver keyword files:
  - **Case B1**: Simulation Model and Loadcase are saved as Solver keyword files
    The Simulation Run contains a copy of the Loadcase and a copy of the Simulation Model, line by line.
  - **Case B2**: Simulation Model and Loadcase are not saved as Solver keyword files
    The Simulation Run contains a copy of the contents of the Simulation Model and the Loadcase, line by line.

If the Simulation Run must contain references to the contents of the Simulation Model and the Loadcase no matter if these intermediate containers are requested to be saved as solver keyword files or not, the option "Reference Inner Contents" can be used (**Case C**), which explicitly requests a reference of the Subsystems and Library Items includes in the Simulation Run main file.

3.8. "Read-only" modules

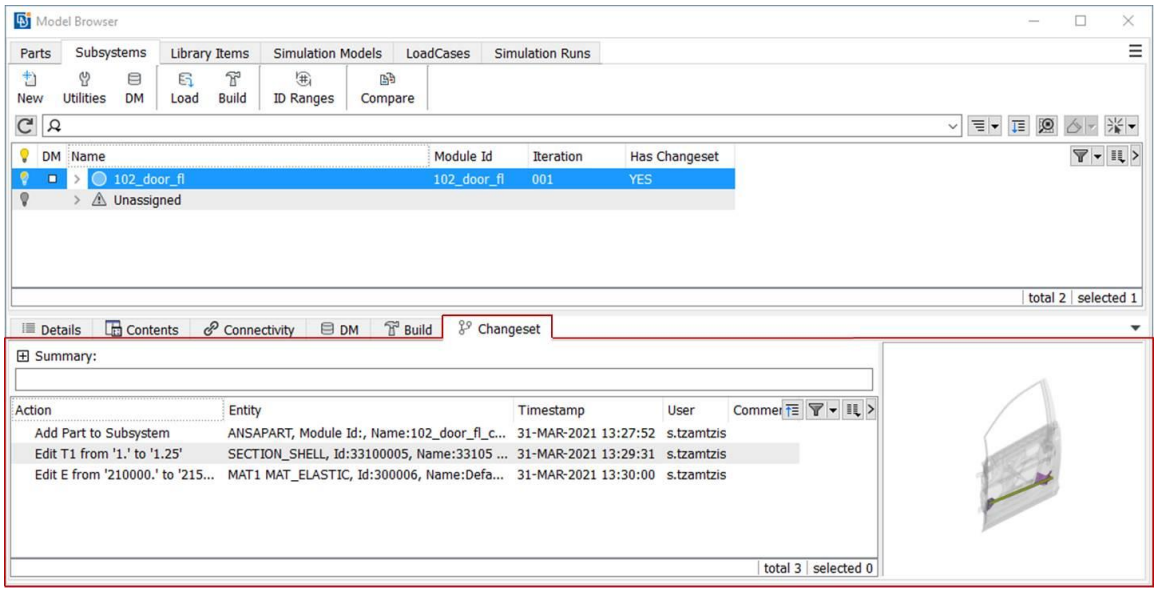
Although the Modular Environment can detect modifications in the contents of base modules in order to mark them for save, there may be cases where a base module needs to be used as a "black box" and must be protected from save. Such handling may be necessary for modules retrieved from suppliers or modules that may contain unsupported keywords or numbering. In order to trigger this special handling, the user can mark a module as "read-only" through the **Is Read Only** attribute in the Model Browser.



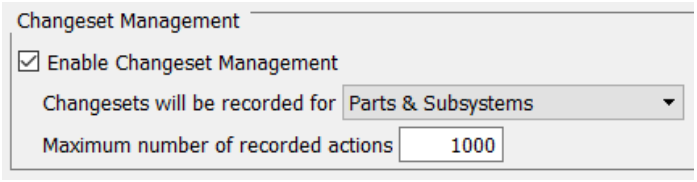
Base modules marked as read-only will be protected from save both when the user selects them and attempts to save them in DM as well as during "silent save".

3.9. Changesets

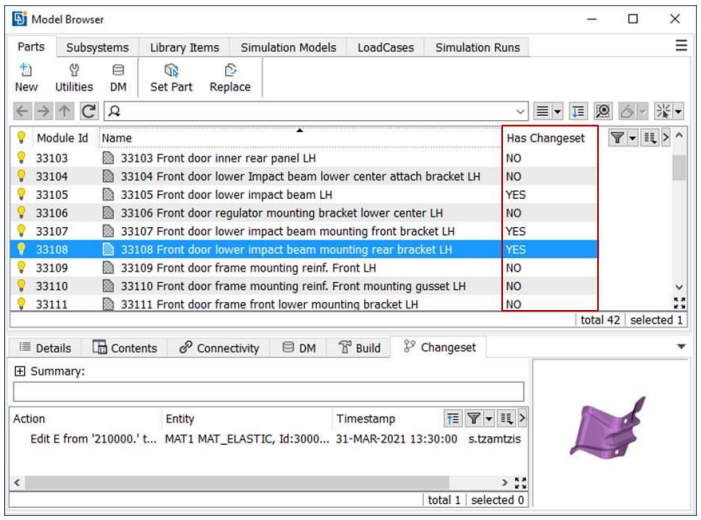
Modifications that change the “DM Update Status” of Parts and Subsystems are recorded and reported as a *changeset*. A changeset represents the *delta* between two successive versions of a Part or Subsystem and is shown in the respective bottom tab in the Model Browser.



The changeset consists of a screenshot that highlights the modified areas plus a list of all the actions that took place since the opening of the file from DM. The *Action* column in the *Changeset* tab displays detailed information for the recorded actions in case of Edit Card field modifications (e.g. the previous and the new value of a Property thickness) or addition/removal of Parts in Subsystems. Information related to the modified entity (e.g. ANSA keyword, Entity Id, Part Module Id, Name) is displayed in the *Entity* column. Moreover, the changeset contains information related to when the modification took place (*Timestamp*), by whom (*User*) and an optional user *Comment*. Finally, the users can also add a *Summary* to describe the engineering intention behind the modifications.



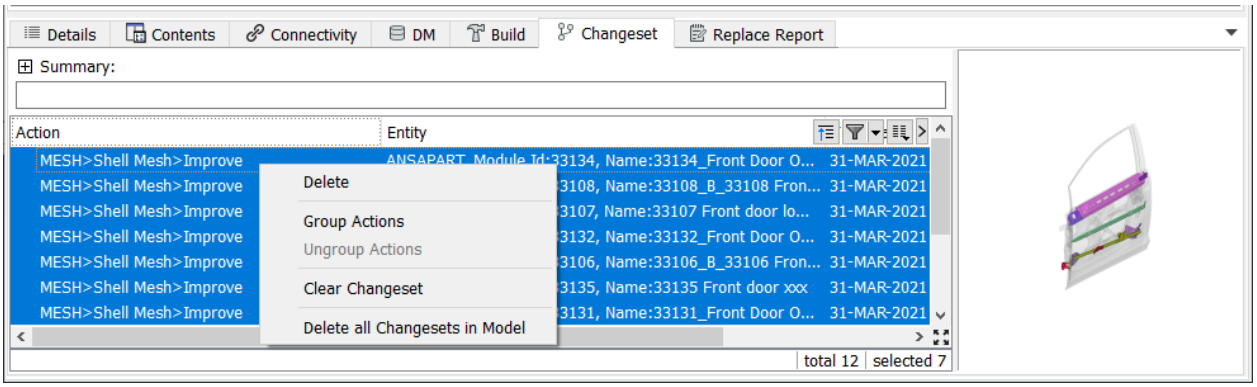
The tracking of modifications as changesets in the model is controlled through the **Enable Changeset Management** setting in the *Changeset Management* section of the *General* tab under *ANSA Settings* > *Model Browser*.



Changesets are recorded for Parts and/or Subsystems. The changesets of Subsystems accumulate the modifications of all the Parts they contain, as well as modifications that relate to their direct contents (Model Setup Entities, Connections, etc.).

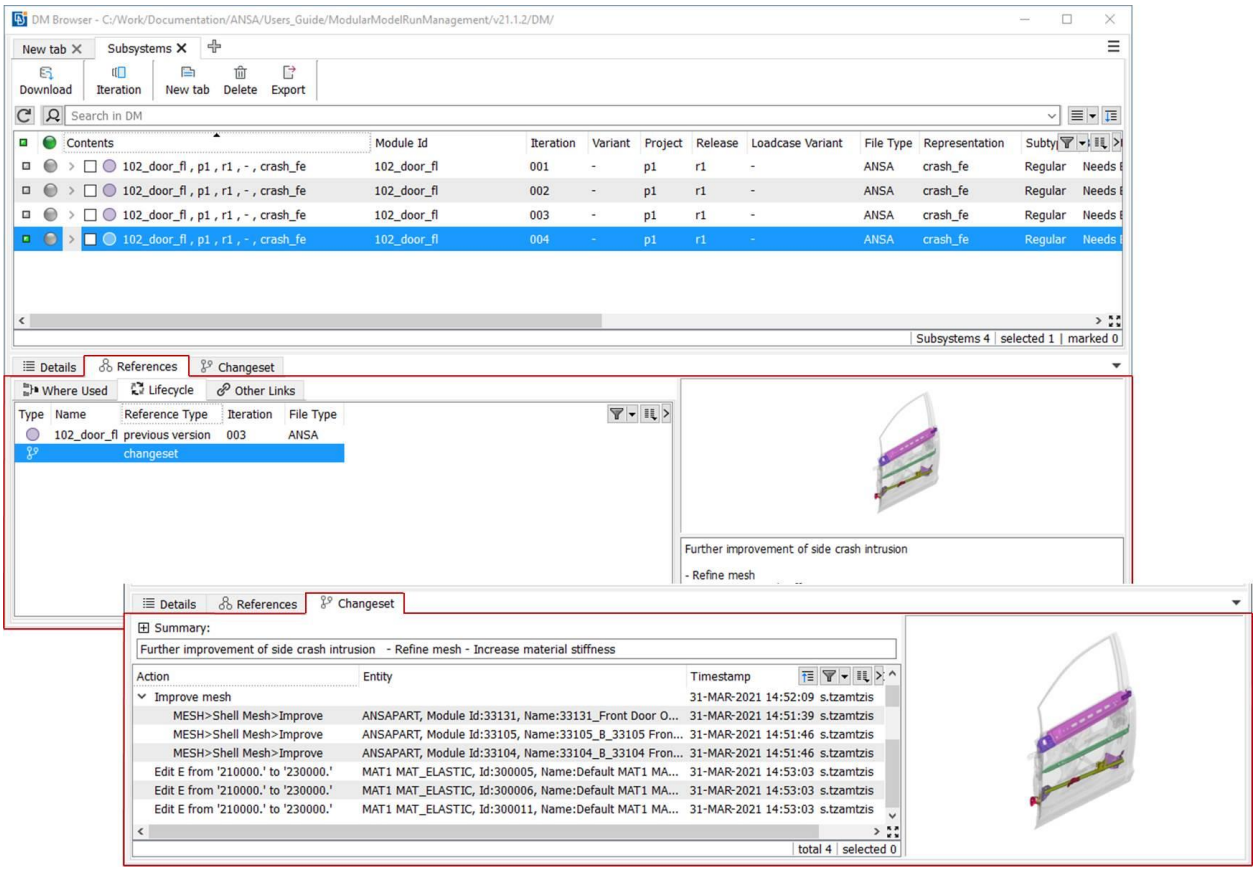
The Parts and Subsystems that have recorded changesets can be identified through the attribute **Has Changeset**.

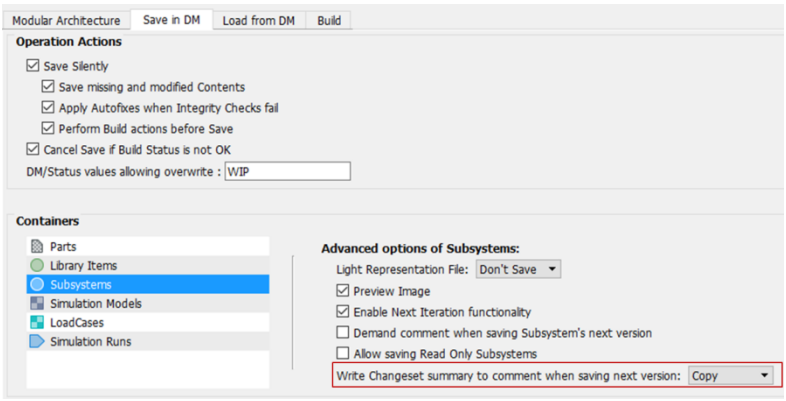
In the *Changeset* tab, n the context menu, several options are available to facilitate the management of the recorded actions. It is possible to select one or more changeset actions and *Delete* them, *Group / Ungroup* a selection of actions, *Clear* the changeset of the selected Part/Subsystem or *Delete all Changesets in Model*.



Changesets are saved in the ANSA file until their associated Model Container is saved in DM. At that moment, the changeset information is cleared from the ANSA file so that a new changeset can be recorded with the next iteration.

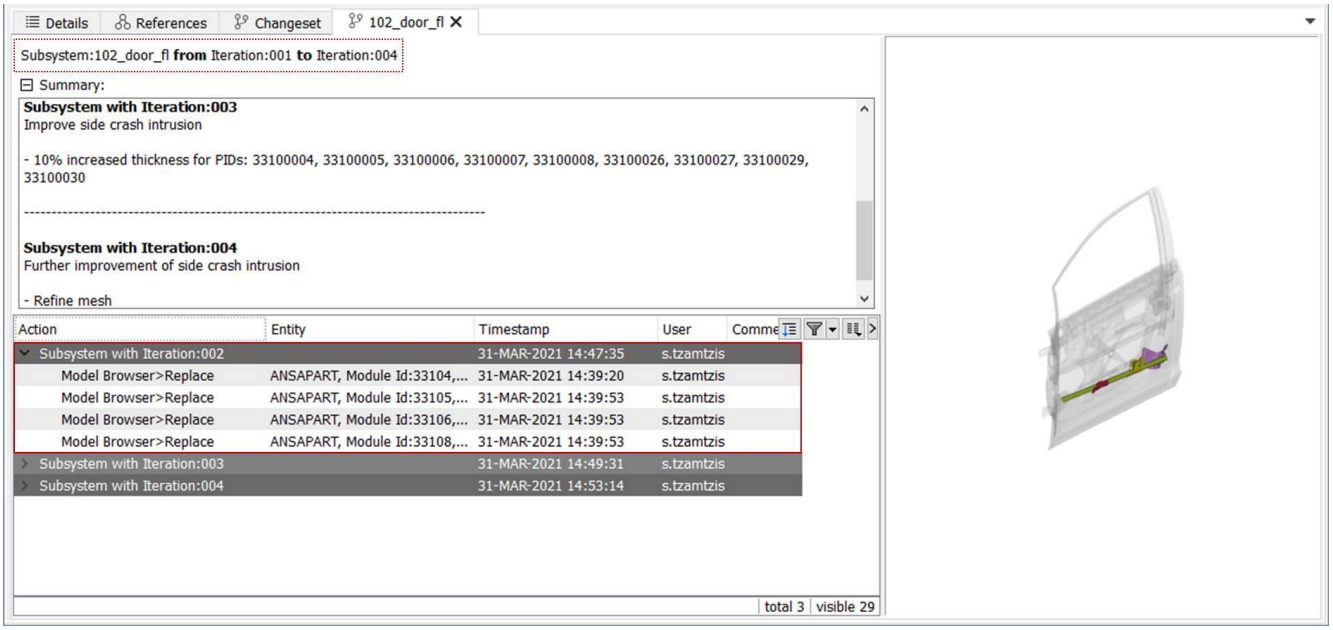
During the saving of the Subsystem in DM, the Changeset is saved along as a separate DM object. In the DM Browser, changesets are only shown as metadata of the Part or Subsystem DM Objects they relate to, in the bottom tabs.





While saving in DM, It is possible to couple the Summary of changesets with the Comment of the Subsystem, so that on Save, the Comment is auto-filled by the changeset Summary automatically. This behavior can be activated through the **Write Changeset summary to comment when saving next versions** setting in the *Save in DM* tab under *ANSA Settings* > *Modular Environment*, as shown on the left.

The complete changeset history between two versions of a Part or Subsystem saved in DM can also be viewed, selecting the desired versions and using the context menu option **Show Changeset history**. A new bottom tab in the DM Browser, labeled after the Module Id of the selected entity, will display all the changesets between the selected versions. A label at the top of the tab will display information for the selected versions; the *Summary* field will display all changeset summaries clearly separated, while the *Actions* are grouped per changeset object. Selecting one of the grouping items, the summary field will be focused on the respective changeset summary and the changeset screenshot will be updated to display the respective image. Finally, the grouping items can be expanded to display the recorded actions for each changeset in detail.



Access to the changeset management functionality is also available through the Python Scripting API, allowing the user to collect information for existing changesets, create new changeset actions or modify the existing, clear or delete the model changesets. It is also possible to collect information of the changeset objects stored in DM, through the DM Objects of the Parts or Subsystems they relate to.

4. Build Process

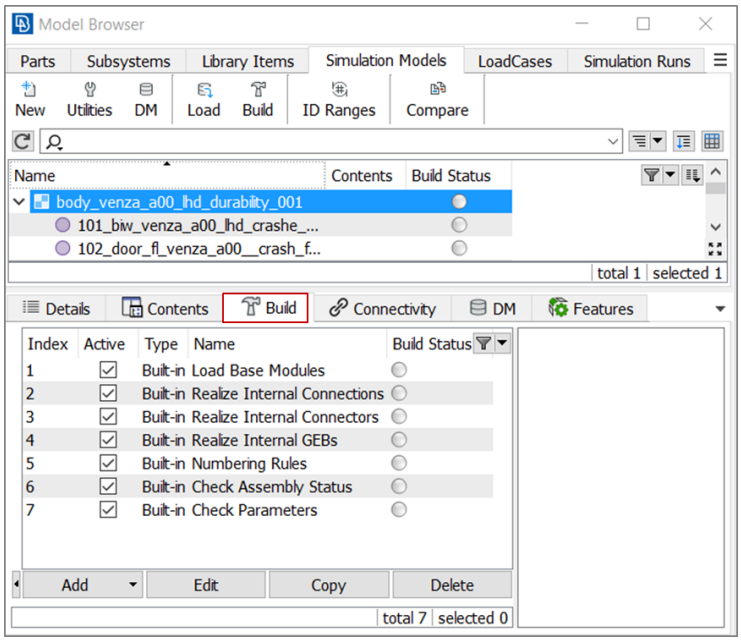
Working in a modular environment where individual users and smaller teams work in parallel on module level, model composition and loadcase setup, it is imperative to ensure the integrity of the final product that will be submitted to the solver to perform an analysis.

In the BETA Suite's Modular Run Environment, each Model Browser Container has a process that must be followed in order to verify that it's ready to be saved in the data repository and it meets the requirements of its parent container for the composition of the model. Theoretically, executing the *Build Process* of a Model Browser Container on its Definition ANSA File, the product of the process is ready to be saved in DM as a solver keyword file.

Having such a process in place offers some key advantages:

- it reduces the reliance on the individual user to persistently follow set rules
- it guarantees the quality of each product (i.e. the keyword file) saved in DM
- it enables the validation of the integrity of the model assembly
- it facilitates the creation of run iterations

The *Build Process* consists of several built-in individual steps, the *Build Actions*, which can be extended and enhanced with the aid of scripting, by adding one or more custom actions. *Build Actions* differ between the various Model Browser Container types (i.e. there are different actions for Subsystems, Library Items, Simulation Models, etc.), while at the same time it is possible to have different Build Actions for each individual Model Browser Container.



The **Build** tab in the *Model Browser* captures the *Build Process* for each Model Browser Container. It lists all the *Build Actions* that are part of the *Build Process* for the selected Model Browser Container, showing the order in which they will be executed, their type and their execution status.

4.1. Built-in Build Actions

The Modular Run Environment comes with pre-defined built-in Build Actions for each Model Browser Container Type, as summarized in the table below:

Build Action Name	Description
Subsystem	
Realize Internal Connections (1)	Realizes all the Connections (spotwelds, adhesives, bolts, seamlines, etc.) that are *internal* to the Subsystem.
Realize Internal Connectors (1) (2)	Realizes all the Connectors that are *internal* to the Subsystem.
Realize Internal GEBs (1) (2)	Realizes all the Generic Entity Builders that are *internal* to the Subsystem. This includes the Assembly Points (A_POINTs), Loadcase Points (LC_POINTs) and Domain Finders.
Source Numbering Rules	Renumbers the Subsystem contents according to the numbering rules defined on the Subsystem.
Check Parameters	Checks that the parameter definition is compatible with the solver of the current deck.
Library Item	
Realize Internal Connections (1)	Realizes all the Connections (spotwelds, adhesives, bolts, seamlines, etc.) that are *internal* to the Library Item.
Realize Internal Connectors (1) (2)	Realizes all the Connectors that are *internal* to the Library Item.
Realize Internal GEBs (1) (2)	Realizes all the Generic Entity Builders that are *internal* to the Library Item. This includes the Assembly Points (A_POINTs), Loadcase Points (LC_POINTs) and Domain Finders.
Source Numbering Rules	Renumbers the Library Item contents according to the numbering rules defined on the Library Item.
Position Impactor (3)	Loads the impactor contained in the Library Item and opens the *Pedestrian Positioning Options* window, where the user needs to set the inputs for the positioning process. A detailed description is given in paragraph 7.3.2.
Check Parameters	Checks that parameter definition is compatible with the current deck solver.
(1) This action will only realize the entities whose Status is not “OK”. (2) For Connecting Subsystems and Loadcase Header Library Items that have been opened standalone, this action will also load the interface representation of the base modules referenced in the connectivity of Connectors / GEBs / Domain Finders, so that they can be re-applied. (3) This action is only available for Target Points Library Items.	

Build Action Name	Description
Simulation Model Adapter	
ID Handling	Checks the validity of the source and target Id ranges definitions. A more detailed description is given in paragraph 5.2.1.
Create Renumbered Deck File (4)	Handles the renumbering of the entities in case the source and target ids are not compatible. A more detailed description is given in paragraph 5.2.1.
Transformations	Handles the transformation of the module based on the transformation file defined under the Transformation File (Auto) attribute. A more detailed description is given in paragraph 5.2.2.
Positioning (5)	Handles the positioning of the module based on the Kinetic Position File and the Kinetic Position defined in the respective attributes. A more detailed description is given in paragraph 5.2.2.
Simulation Model	
Load Base Modules	For the empty (unloaded) contents of the Simulation Model, it loads the interface representation file for Regular Subsystems and Library Items and the ANSA definition file for Connecting Subsystems. In addition, it executes the build process of Connecting Subsystems and realizes the *intermodular* Connectors that are assigned directly to the Simulation Model (i.e. entities contained in the Connections container of the Simulation Model). Finally, it executes the build process of Loadcase_Header or Target_Points Library Items.
Realize Internal Connections (1)	Realizes all the Connections (spotwelds, adhesives, bolts, seamlines, etc.) that are assigned to the Simulation Model.
Realize Internal Connectors (1)	Realizes all the Connectors that are assigned to the Simulation Model.
Realize Internal GEBs (1)	Realizes all the Generic Entity Builders that are assigned to the Simulation Model.
Numbering Rules	Renumbers the entities assigned directly to the Simulation Model (entities contained in the Connections and Model Setup Entities containers of the Simulation Model) according to the numbering rules defined on the Simulation Model.
Check Assembly Status	Checks that no undefined interface point exists and that all interface points have Status “OK”.
Check Parameters	Checks that parameter definition is compatible with the current deck solver.

(1) This action will only realize the entities whose Status is not “OK”.
(4) This action is not available in LsDyna and Radioss.
(5) This action is only available in LsDyna, PamCrash, Abaqus and Radioss.

Build Action Name	Description
Loadcase Adapter	
ID Handling	Checks the validity of the source and target Id ranges definitions. A more detailed description is given in paragraph 5.2.1.
Create Renumbered Deck File (4)	Handles the renumbering of the entities in case the source and target ids are not compatible. A more detailed description is given in paragraph 5.2.1.
Transformations	Handles the transformation of the adapter based on the transformation file defined under the **Transformation File (Auto)** attribute. A more detailed description is given in paragraph 5.2.2.
Positioning (5)	Handles the positioning of the adapter based on the **Kinetic Position File** and the **Kinetic Position** defined in the respective attributes. A more detailed description is given in paragraph 5.2.2.
Create Simulation Runs (6)	Opens the *Create Simulation Runs* window where the analyst can select the Target Points for which a new Simulation Run must be created. A detailed description is given in paragraph 7.3.3.
Loadcase	
Load Base Modules	For the empty (unloaded) contents of the Loadcase, it loads the interface representation file for Regular Subsystems and Library Items and the ANSA definition file for Connecting Subsystems. In addition, it executes the build process of Connecting Subsystems and realizes the *intermodular* Connectors that are assigned directly to the Loadcase (i.e. entities contained in the Connections container of the Loadcase). Finally, it executes the build process of Loadcase_Header or Target_Points Library Items.
Realize Internal Connections (1)	Realizes all the Connections (spotwelds, adhesives, bolts, seamlines, etc.) that are assigned to the Loadcase.
Realize Internal Connectors (1)	Realizes all the Connectors that are assigned to the Loadcase.
Realize Internal GEBs (1)	Realizes all the Generic Entity Builders that are assigned to the Loadcase.
Numbering Rules	Renumbers the entities assigned directly to the Loadcase (entities contained in the Loadcase Connections and Model Setup Entities containers) according to the numbering rules defined on the Loadcase.
Check Assembly Status	Checks that no undefined interface point exists and that all interface points have Status “OK”.
Check Parameters	Checks that parameter definition is compatible with the current deck solver.

(1) This action will only realize the entities whose Status is not “OK”.
(4) This action is not available in LsDyna and Radioss.
(5) This action is only available in LsDyna, PamCrash, Abaqus and Radioss.
(6) This action is only available for the adapters of Target Points Library Items.

Build Action Name	Description
Simulation Run Adapter	
ID Handling	Checks the validity of the source and target Id ranges definitions. A more detailed description is given in paragraph 5.2.1.
Create Renumbered Deck File (4)	Handles the renumbering of the entities in case the source ids are not compatible with the target ids. A more detailed description is given in paragraph 5.2.1.
Transformations	Handles the transformation of the adapter based on the transformation file defined under the Transformation File(Auto) attribute. A more detailed description is given in paragraph 5.2.2.
Simulation Run	
Check Interface Points Uniqueness	For META FRF ASSEMBLY Loadcases, it checks if Loadcase Points and Assembly Points have unique names.
Check Parameters	Checks that: No parameters associated with the Simulation Run remain undefined. Parameter definition is compatible with the current deck solver.
(4) This action is not available in LsDyna and Radioss.	

It should be noted that during the execution of the *Build Process* for Connecting Subsystems and Loadcase Header Library Items, the application status of Domain Finders and GEBs is "protected" by ANSA in order to avoid the unnecessary creation of new versions of the base modules. The search result of Domain Finders and GEBs is evaluated so that their Status will be reset only if any changes are detected based on the model content. This will also reset the *Build Status* of their parent Model container, triggering the execution of the *Realize Internal GEBs Build Action* that would lead to their re-application.

A characteristic example for this would be the case of a Connecting Subsystem that contains a Domain Finder searching in the model for Interface Sets. When the first iteration of the model is saved in DM, the Domain Finder is applied and the collected Interface Sets are added to the Connecting Subsystem. During the creation of the next iteration of the model, ANSA will evaluate if the search of the Domain Finder would lead to different Interface Sets being collected compared to the existing result in the Connecting Subsystem. If no changes are detected, the same Connecting Subsystem can be re-used in the next model iteration. On the other hand, if changes are detected, the Domain Finder status will be reset, also resetting the Build Status of the Simulation Model. The *Realize Internal GEBs Build Action* can then be executed, in order to re-apply the Domain Finder and save a new version of the Connecting Subsystem in DM.

Moreover, ANSA offers specific handling for the *Build Process* of base modules (i.e. Subsystems and Library Items) that contain unsupported keywords or unsupported numbering. As described in paragraph 3.8, such modules can be marked as read-only. When the *Build Process* needs to adapt the content of a read-only base module, it will always retrieve its definition attributes from DM, even in cases where it is fully loaded in ANSA. A more detailed description of the adaptation of loaded entities can be found in paragraph 5.3.

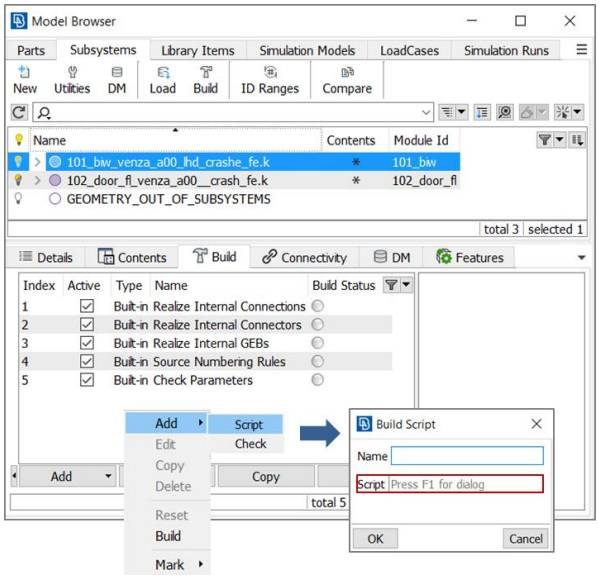
Finally, when it comes to the *Build Process* of Simulation Models and Loadcases, ANSA by default offers the same *Build Actions* regardless of the Model Browser Container content. This means that the same *Build Actions* that will be executed for Library Items and for Subsystems, through the Simulation Model Adapter or the Loadcase Adapter.

4.2. Customization of the Build Process

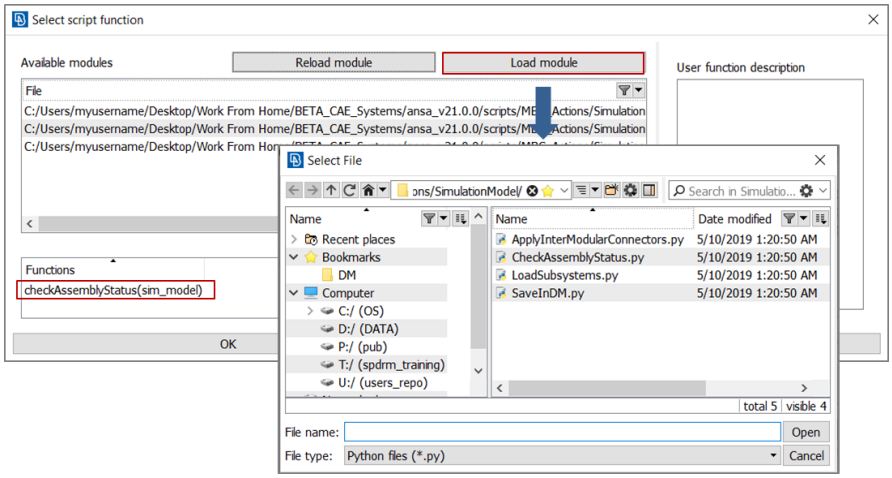
To support the various processes employed by CAE teams, the Modular Run Environment allows the customization of the *Build* Process. It is possible to add to the built-in *Build Actions by* defining custom actions, while at the same time is possible to define different actions for each individual Model Browser Container of the same type. In addition, it is possible to control which actions will be active when the *Build Process* of a Model Browser Container is executed, through the respective **Active** setting in the *Model Browser* **Build** tab.

4.2.1. Script Build Actions

Custom *Build Actions* can be defined with the aid of scripting through the **Build** tab of the *Model Browser*. Using the option **Add > Script** from the context menu of the **Build** tab, a script build action can be added to the *Build Process* of a Model Browser Container. The definition of the script build action requires the assignment of a *Name* and the selection of a *Script* file in the *Build Script* window.



Pressing the *F1* button on the keyboard in the *Script* field, the *Select script function* window opens which lists any script files already loaded in the current ANSA session. Using the **Load module** option, it is possible to select any script file from the file system. Upon selecting any of the available script files listed in the *Select script function* window, the available script functions that match the required function definition prototype are listed and can be selected as the main function to be executed when the script build action is executed.



The function definition prototype for *Script Build Actions* requires a single input argument that will correspond to the Model Browser Container being built. The expected return value is an integer that will be interpreted as an indication of success or failure (1 for success and 0 for failure). A code snippet like the one shown below could check the status of the assembly points of a Subsystem.

Code snippet

```
import ansa
from ansa import base

def checkInterfacePointsStatus(mbcontainer):
      if (mbcontainer.ansa_type(0)) == "ANSA_SIM_MODEL_ADAPTER":
            print("Adapters do not contain interface points. Build Action skipped!")
            return 1
      interfaces= base.CollectEntities (0, mbcontainer, "A_POINT")
      if not len(interfaces):
            print ("No interface points found. Nothing to do!")
            return 1
      is_err = False
      undefined_interfaces = []
      unapplied_interfaces = []
      for a_point in interfaces:
            p_info = a_point.get_entity_values(0,('DEFINED', 'status', 'Interface
Node','MBContainer'))
            if p_info['DEFINED'] == 'NO':
                  undefined_interfaces.append(a_point)
                   is_err = True
            else:
                  if p_info['status'] != 'ok':
                        if not (p_info['status'] == '' and p_info['Interface Node']
== 0):
                              pass
                        else:
                              unapplied_interfaces.append(a_point)
                              is_err = True
      if is_err:
            print ("The Build function 'Check Interface Points Status' has
identified an error in the Assembly and the process will stop")
       return 0
      return 1
```

Output

If defined on a Simulation Model adapter rather than a Subsystem, the Build Action will be skipped (considered a successful completion) and the following will be printed:
"Adapters do not contain interface points. Build Action skipped!"
If not interface points are found in the Subsystem, the Build Action will be skipped (considered a successful completion) and the following will be printed:
"No interface points found. Nothing to do!"
If any interface points with Status other than "ok' exist in the Subsystem, the Build Action will fail and the following will be printed:
"The Build function 'Check Interface Points Status' has identified an error in the Assembly and the process will stop"
If all interface points are well defined and applied, the Build Action will be successfully completed and no message will be printed.

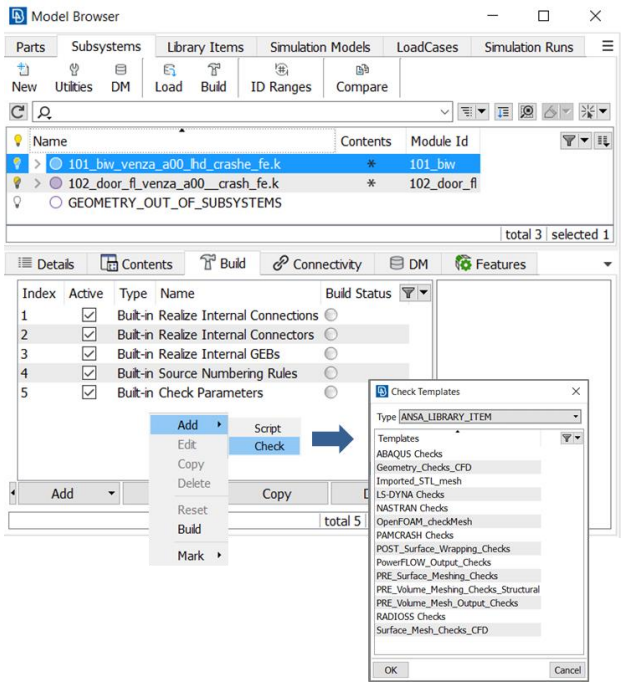
Newly defined custom *Build Actions* are listed at the end of the build actions list in the *Build* tab. When a script *Build Action* is selected, the lower right-hand side section of the *Build* tab displays the file path and docstring of the script and the script function name that will be executed with the build action.

Script build actions defined through the Model Browser *Build Tab* are only valid for the Model Browser Container for which they were created. In order for all new Model Browser Containers to have the same custom *Build Process*, the definition of custom *Build Actions* must be given in the ANSA defaults, as described in section 4.2.3

Note ! It is possible to rearrange the order of the *Build Actions* in the *Build* tab using drag and drop operations.

4.2.2. Check Build Actions

Apart from custom *Build Actions* with the aid of scripting, it is also possible to define custom actions with the aid of ANSA Check Templates. Using the option **Add > Check** from the context menu of the **Build** tab, a check build action can be added to the *Build Process* of a Model Browser Container. In the *Add Check* window the available Check Templates are listed and can be selected as the checks to be executed when the check build action is executed. The newly defined *Check Build Action* will be listed at the end of the build actions list in the **Build** tab.



Check build actions defined through the *Model Browser* **Build** Tab are only valid for the Model Browser Container for which they were created. In order for all new Model Browser Containers to have the same custom *Build Process*, the definition of custom *Build Actions* must be given in the ANSA defaults, as described in section 4.3.3

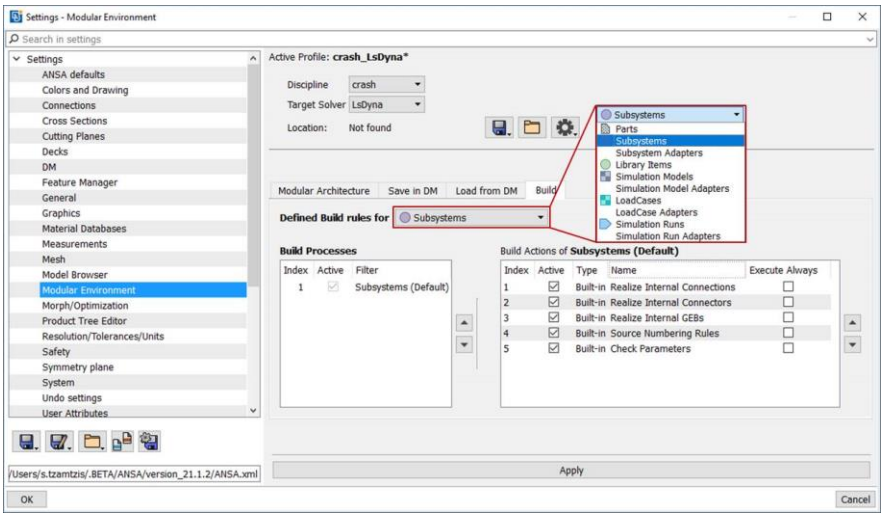
Note ! It is possible to rearrange the order of the *Build Actions* in the **Build** tab using drag and drop operations.

4.2.3. Controlling the content of the Build Process

The *Build Process* of the various Model Browser Containers can be controlled on two levels:

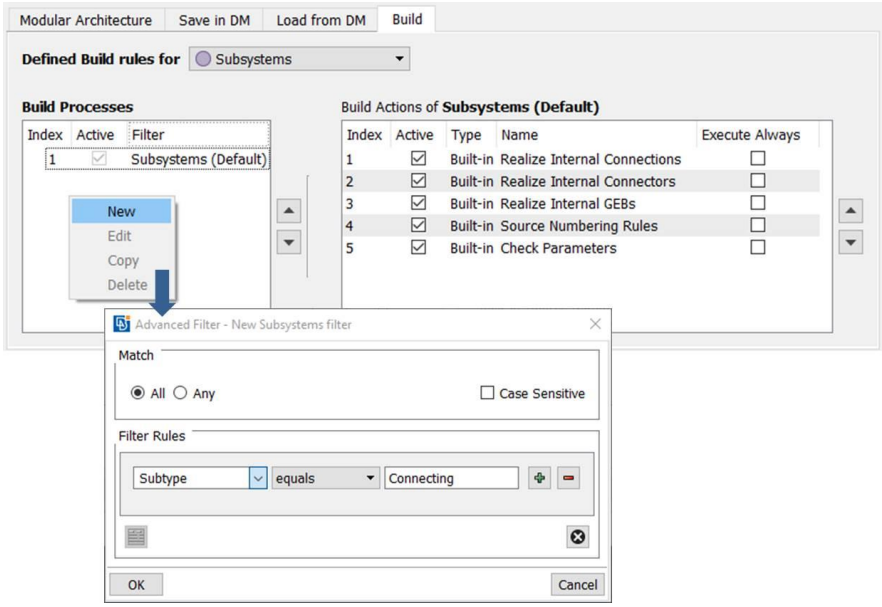
- In the Modular Environment Profile settings, it is possible to define the default process for each type of Model Browser Container or differentiate the *Build Process* for containers of the same type using filters.
- In the ANSA database, it is possible to control which actions will be active when the *Build Process* will be executed for each individual Model Browser container.

The former can be used to ensure that the same *Build Process* is defined for all users within a team and will be executed to verify that the Model Browser Containers meet a set of requirements for the modular model composition, while the latter allows for further customization of the *Build Process* for individual containers in the model.



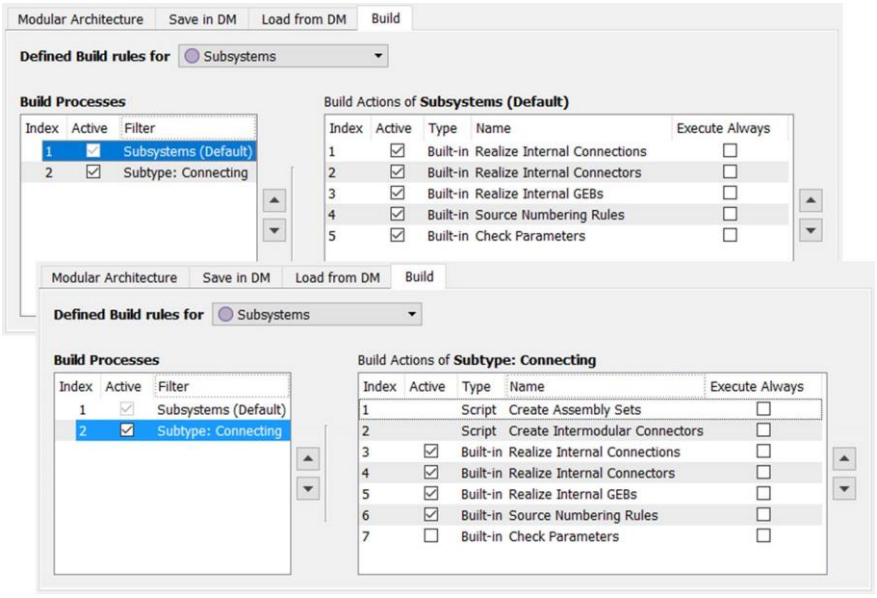
A default *Build Process* for each type of Model Browser Container is defined through the ANSA defaults.

In the **Build** tab in the **ANSA Settings > Modular Environment**, the *Build Actions* that will be available for each type of Model Browser Container are controlled, as well as the order in which they will be executed. These may consist of a combination of built-in and custom actions (*Script* and/or *Check Build Actions*).

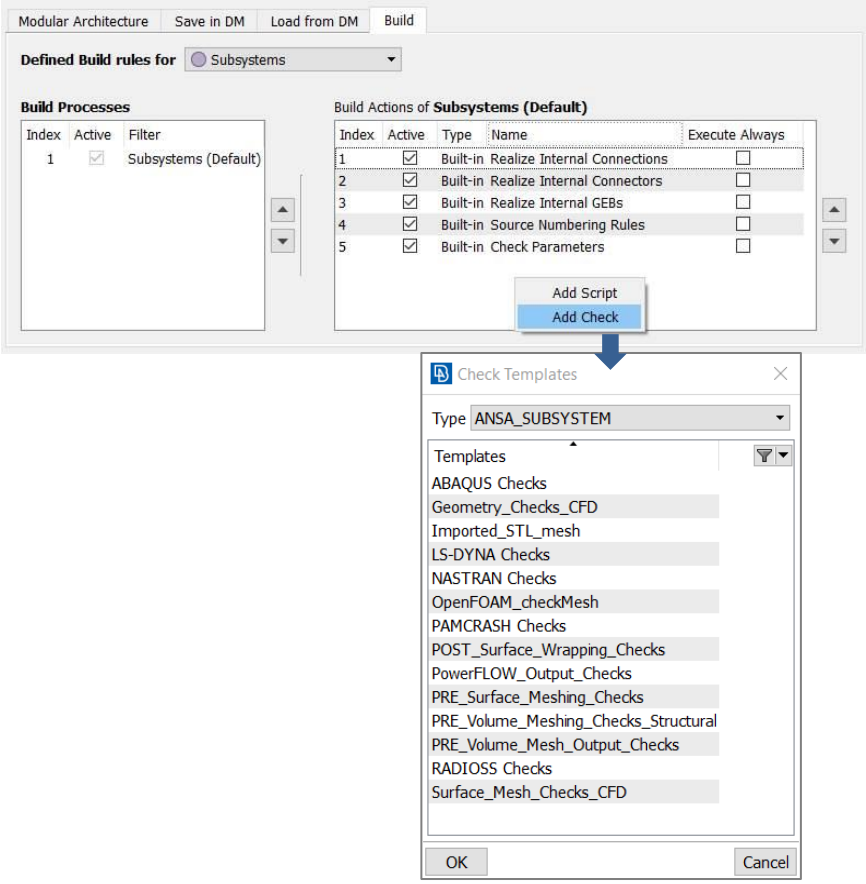


It is also possible to further differentiate the *Build Process* for containers of the same type.

Using the **New** option from the context menu, advanced filter rules can be defined to create different *Build Processes* for Model Browser Containers of the same type that satisfy the filter rules.

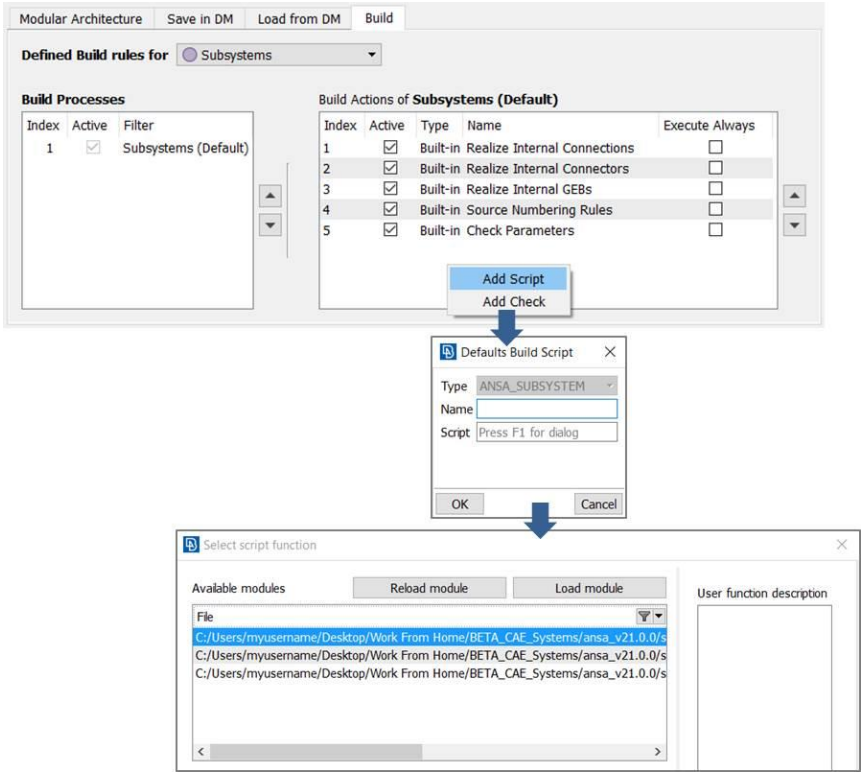


For example, it is possible to define an advanced filter based on the Subsystem Subtype, in order to differentiate the *Build Process* between *Regular* and *Connecting* Subsystems.



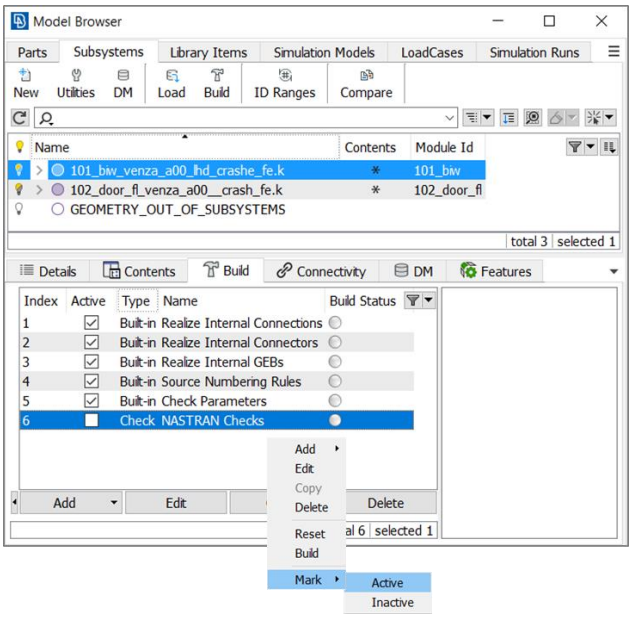
Using the **Add Check** option from the context menu, custom check actions that will be available by default in the *Build Process* of each type of Model Browser Container can be defined.

In the *Check Templates* window that opens, check templates can be selected for the selected Model Browser Container type and added in the default *Build Process*.



Custom script actions that will be available by default in the *Build Process* of each type of Model Browser Container can also be defined.

Using the **Add Script** option, the *Defaults Build Script* window opens where the script that will be used by the custom build action can be defined. Pressing the **F1** key in the Script field opens the *Select script function* window, where the script can be loaded and the script function that will be executed with the custom build action can be selected.



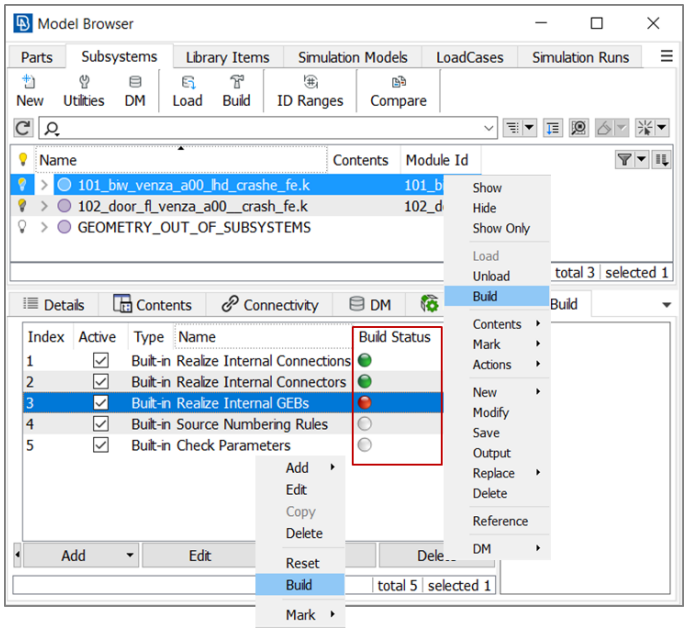
The *Build Process* can be further customized for each individual Model Browser Container, by controlling which of its default actions will be active when the *Build Process* will be executed, or by adding custom check or script actions.

To define the active *Build Actions*, either the respective "Active" checkboxes in the Model Browser *Build* tab or the context menu option **Mark > Active/Inactive** can be used. When the build process is executed, only the actions marked as "Active" will be applied.

Using the options **Add > Script / Check**, custom actions can be added to the *Build Process* of the selected Model Browser Container, as described in paragraphs 4.2.1 and 4.2.2.

4.3. Build Process execution

The execution of a *Build Process* can ensure the integrity of the Model Browser Container that will be saved in DM. Starting from the definition file of a module (i.e. the .ansa file), a series of actions can be applied, such as the application of its connections and connectors, the renumbering and positioning of its contents, the execution of some checks, etc., that will transform this definition file to the final, ready-to-use product (i.e. the solver keyword file). This build product can then be saved in DM and used in a simulation.

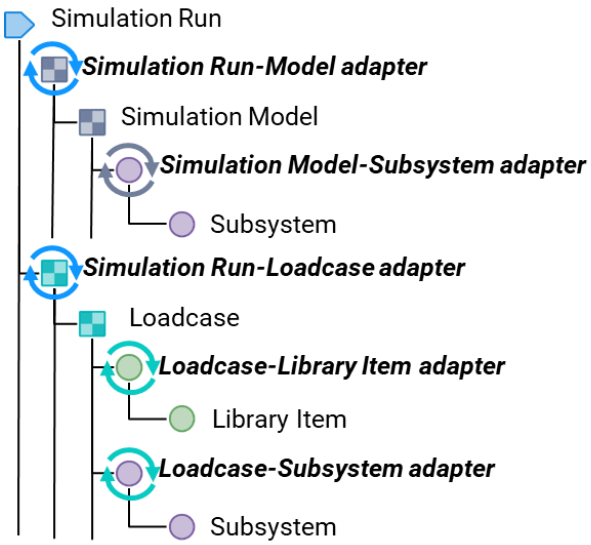


The execution of the *Build Process* is initiated from the **Build** option in the context menu of a Model Browser Container. Alternatively, it is possible to execute an individual *Build Action* using the respective option from its context menu in the *Build* tab. Each action has a status that will either be "Needs Build" (grey) before it's executed, or "OK" (green) in case it is executed successfully or "Error" (red) in case its execution fails. This can be seen in the "Build Status" column in the *Build* tab.

The **Build Status** of a Model Browser Container is displayed in the respective attribute that can be seen in the *Details* tab of the Model Browser. Before executing its *Build Process*, the *Build Status* of a Model Browser container is "Needs Build". The execution is sequential in the order defined in the *Build* tab and it is required that a *Build Action* is successfully completed before then next action can be executed. If any of the contained *Build Actions* fail, the *Build Process* is halted with "Error" *Build Status* for the Model Browser Container. When all *Build Actions* are executed successfully, the Model Browser Container gets *Build Status* "OK". The execution of each *Build Action* depends on its status. Actions whose status is "OK" are not executed again, unless they are marked as "Execute Always" using the characteristic option in the *Build* tab. It should be noted that the "Create Simulation Runs" Build Action of the Loadcase Adapters of Target Points Library Items is marked as "Execute Always" by default.

For Subsystems and Library Items, the *Build Process* consists of the execution of their active *Build Actions* defined in the *Build* tab. For compound Model Browser Containers (i.e. Simulation Models, Loadcases and Simulation Runs) on the other hand, the *Build Process* consists of the following steps:

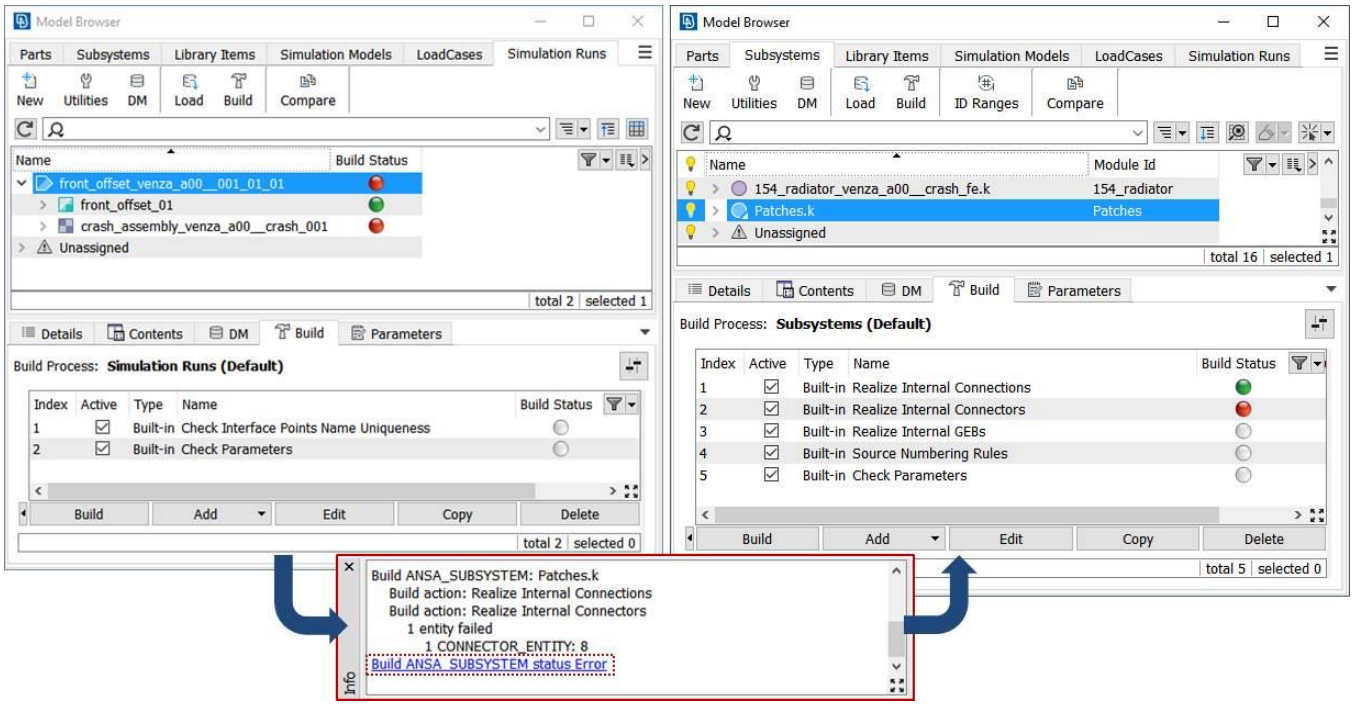
1. The contained Model Browser Containers are built first.
2. The adapters of the compound Model Browser Container are built second.
3. The compound Model Browser Containers are built last.



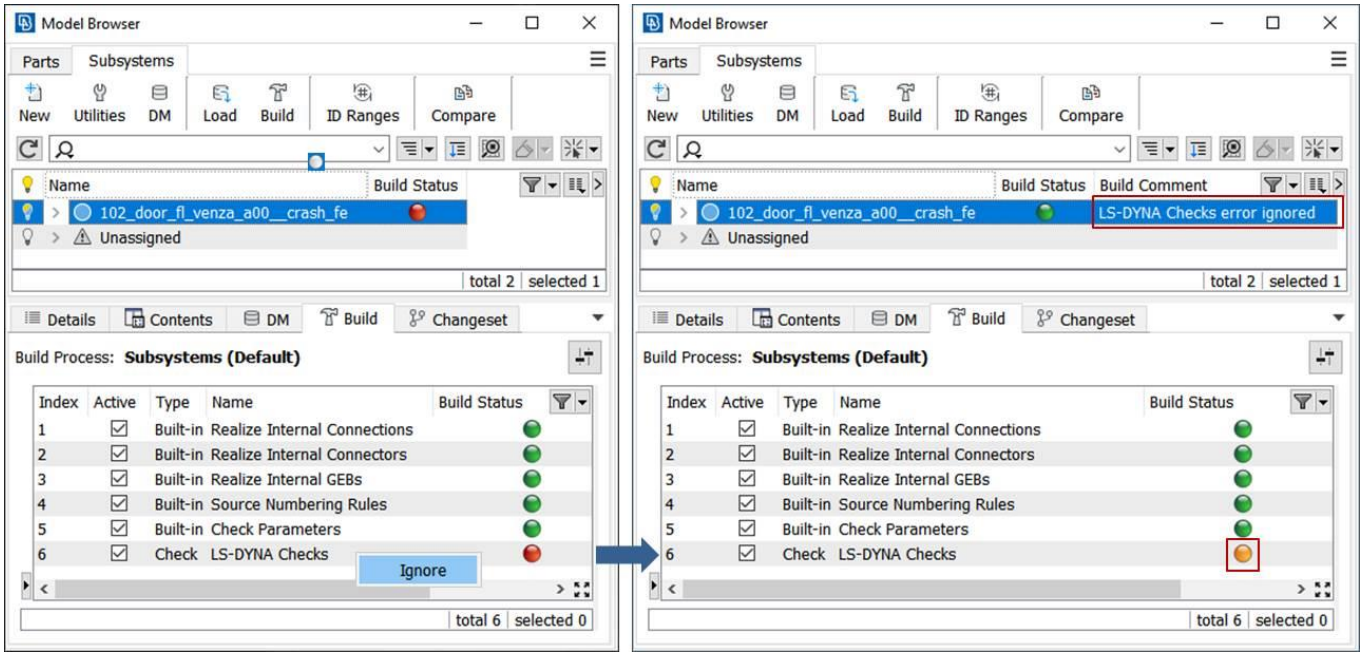
In the structure shown here, when the *Build Process* of the Simulation Run is executed, ANSA will execute all associated *Build Actions* bottom up. Therefore, the *Build Process* of the Subsystems and Library Items will be executed first, followed by the *Build Process* of the Simulation Model Adapters and Loadcase Adapters. Then the *Build Process* of the Simulation Model and the Loadcase will be executed, followed by the *Build Process* of the Simulation Run Adapters. Finally, the *Build Process* of the Simulation Run will be executed. In case that any of the *Build Actions* in the intermediate steps fail, the *Build Process* stops with "Error" *Build Status* for the failed Model Browser Container, which will be propagated to all higher-level Model Browser Containers too.

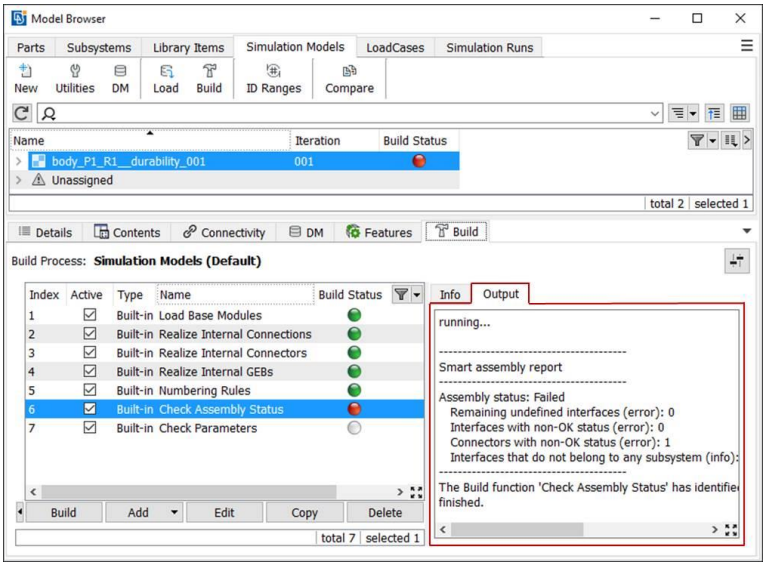
Note ! *Build Actions* can be executed on either loaded or unloaded Model Browser Containers. When working with unloaded containers, the *Build Actions* on the base modules just pass and it is the *Build Actions* defined on adapters and compound Model Browser containers that are executed, automatically loading any required entities.

When the *Build Process* of a Compound Model Browser Container is executed and an error is identified in the *Build Process* of one of its contents, a hyperlink is created in the *ANSA Info* window that redirects the user to the Model Browser tab where the problematic Model Container is listed and selects it automatically, in order to assist the user in the identification of the problem.



Failed *Build Actions* can be bypassed using the **Ignore** option from the context menu. Selecting this option, which is only available for *Build Actions* with “Error” *Build Status*, will switch the status of the *Build Action* to "Ignored", allowing the *Build Process* to continue. This action will be recorded in the overall *Build Comment* of the Model Container. As soon as the *Build Status* of the container is reset, the "Ignored" status will be removed and the *Build Action* will be normally executed the next time the *Build Process* is triggered.





When the *Build Process* of a Model Browser Container is executed, the output of each Build Action is printed in the ANSA info window, among other of the current session. To ensure that the relevant information is saved for each Build Action, the output messages of each Build Action are also stored in the ANSA file and are accessible through the "Output" tab on the bottom right hand side of the Build tab.

The behavior of the *Build Process* can be affected by some of the *Build* Attributes of the Model Browser Container. In particular:

- The *Id Handling* attribute of the adapters controls whether the execution of the respective *Build Action* will only validate the Ids or if the Model Browser Container will be renumbered. A detailed description of this behavior is given in paragraph 5.2.1.
- The *Impactor Type* attributes controls which impactor will be selected for the creation of the Simulation Runs from a Target Points Loadcase. A detailed description of this is given in paragraph 7.3.

Finally, the execution of the *Build Process* can be fully integrated with the save in DM process. It is possible to set up ANSA in such a way that the *Build Process* is automatically executed before saving a Model Browser Container in DM and that the save operation is cancelled in case the *Build Process* is not successfully completed. This is achieved through the “Silent Save” settings, described in detail in paragraph 3.7.

5. Adaptation

The term “adaptation” refers to the process of transforming a module so that it fits the specifications defined by its “consumer” compound container. Three common “adaptation” examples are given below:

- For a particular Simulation Model (i.e. consumer), a Subsystem (i.e. module) that represents a module of the left side needs to be reused on the right side of the assembly
- For a particular Loadcase (i.e. consumer), the same crash dummy (i.e. module) needs to be used as both driver and passenger
- Two Loadcases (i.e. consumers) need to use the same Library Item (i.e. module) with control cards, using a different termination time in each

What all three examples have in common is that typically, in order to use the same module with different specifications, one would have to create as many copies of the module file as the number of different specifications and “adapt” each copy accordingly. However, the creation of file copies, apart from increasing the storage volume requirements, is a potentially problematic practice since the copies are disconnected from each other and from the source file and traceability is lost.

Acknowledging the challenge, several solvers have provided solutions that enable the “adaptation” of a module without the need for a file copy. Some characteristic examples are:

- Parameter keywords: The use of parameters enables the parameterization of a module, so that some of its characteristics can be controlled “from the outside”, by applying the instructions given on “consumer” level
- Spatial transformation keywords: The use of transformation keywords enables the application of a geometric transformation on a module
- ID transformation keywords: It is possible to control the target id range of a module by defining id offset values (positive or negative numbers) which are finally added to the source ids

The implementation of “adaptation” in the Modular Environment is done in a way that exploits the solutions offered by the solvers at the maximum extent and, for all those cases where file copies cannot be avoided, makes sure that the file copies don’t lose track of the source files and always “know” with which specifications they were created. Process-wise, the adaptation is managed through the Build Process. The Modular Environment comes with a set of built-in Build Actions that aim at the application of adaptation.

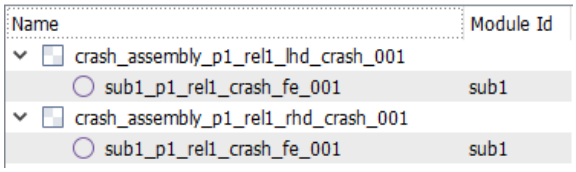
5.1. Adapters

In the Modular Environment, in order to be able to reuse the same Subsystem (e.g. a wheel) in different locations under the same Simulation Model or reuse the same crash dummy Library Item as both a driver and a passenger in the same Loadcase or reuse the same control cards Library Item under two Loadcases with different termination time, one needs to make use of the *adapters*. The adapters in ANSA are created the moment a Model Browser Container is added to a compound container. There are 4 different adapter types supported, one for each of the four compound Model Browser Container types:

- ANSA_SUBSYSTEM_GROUP_ADAPTER: Used to link Subsystem Groups with their contents
- ANSA_SIM_MODEL_ADAPTER: Used to link Simulation Models with their contents
- ANSA_LOADCASE_ADAPTER: Used to link Loadcases with their contents
- ANSA_SIM_RUN_ADAPTER: Used to link Simulation Runs with their contents

Although the adapter entity types are not displayed in the Database Browser, they can be managed through script similarly to all other ANSA entities. They have an edit card that can be accessed with double click on the *adapter* in

the Model Browser. In the Model Browser, the term *adapter* refers to a child entry in the lists of compound Model Browser Containers. For example, when one selects a Subsystem by expanding a Simulation Model in the list of Simulation Models, he essentially selects the *adapter* of this Subsystem.



In a Model Configuration Table, where the composition of different Simulation Model configurations is defined, it's very common to find the same Subsystem under different Simulation Models. In the example below, subsystem with name "sub1_p1_rel1_crash_fe_001" is used under 2 different Simulation Models that represent the right-hand drive and the left-hand drive variants of the complete assembly.

A code snippet like the one shown below could identify quickly all *adapters* starting from the Subsystem.

Code snippet

```
import ansa
from ansa import base
get_ent = base.NameToEnts('sub1_p1_rel1_crash_fe_001')
sub_ent = get_ent[0]
sim_models = base.CollectEntities (0, None, 'ANSA_SIMULATION_MODEL')
for sim_model_ent in sim_models:
   smodel_adptr = base.GetModelContainerAdapter(sim_model_ent, sub_ent)
   print (smodel_adptr.ansa_type(0))
```

Output

'ANSA_SIM_MODEL_ADAPTER'
'ANSA_SIM_MODEL_ADAPTER'

In order to collect adapters top-down (i.e. from a Simulation Model get its Simulation Model Adapters), base.CollectEntities can be used, using the Simulation Model as a container.

Code snippet

```
import ansa
from ansa import base

sim_models = base.CollectEntities (0, None, 'ANSA_SIMULATION_MODEL')
for sim_model_ent in sim_models:
   smodel_adapters = base.CollectEntities(0, sim_model_ent, 'ANSA_SIM_MODEL_ADAPTER')
   for one_adptr in smodel_adapters:
      adptr_info = one_adptr.get_entity_values(0, ['Child ID', 'Child type', 'Parent ID',
'ID'])
      print ('Adapter with id %s adapts %s with id %s under Simulation Model with id
%s'%(adptr_info ['ID'], adptr_info['Child type'], adptr_info['Child ID'], adptr_info['Parent
ID']))
```

Output

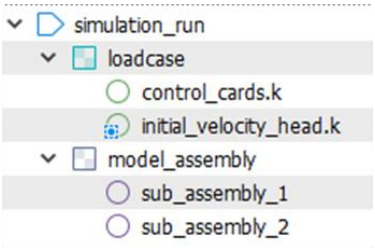
Adapter with id 1 adapts ANSA_SUBSYSTEM with id 2 under Simulation Model with id 4
Adapter with id 2 adapts ANSA_SUBSYSTEM with id 2 under Simulation Model with id 6

5.2. Adapting attributes and adaptation process

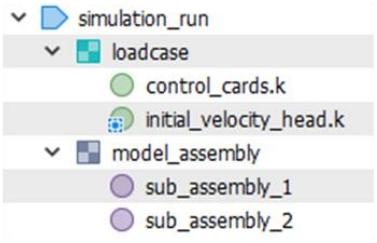
The adapters are used as placeholders of attributes that concern the use of a Model Browser Container within its parent. These attributes, that control everything “composition-related”, from the order with which the children containers will be written out into the main keyword file of their parent to the geometric transformation that will be used to position a child container into its parent, are either defined through special tools or manually.

Once the adapting attributes are defined, the adaptation process can run which, based on the attributes, will generate the required metafiles that represent the adapted child container.

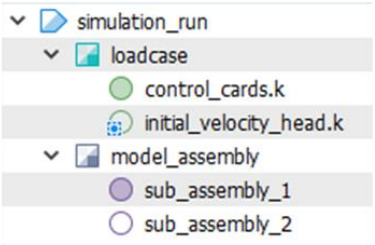
The adaptation process is applicable on both loaded and unloaded Model Browser Containers. This means that the required metafiles of the adapted data can be produced either from an ANSA session where only the *Definition File* of the compound Model Browser Container is loaded or from a session where all Model Browser Containers are loaded with their FE entities. Some peculiarities of the process when working with loaded data are described in paragraph 5.3.



Unloaded Model Browser Containers



Loaded Model Browser Containers



Partially loaded Model Browser Containers

At present, ANSA supports four main adaptation processes in order to control the following characteristics of modular models:

1. The id ranges of the different modules
2. The geometric position of the different modules
3. The values of parameters of parameterized modules
4. The order with which INCLUDE keywords appear within a main keyword file

The adaptation processes 1 through 3 require access to the source attributes of a subsystem

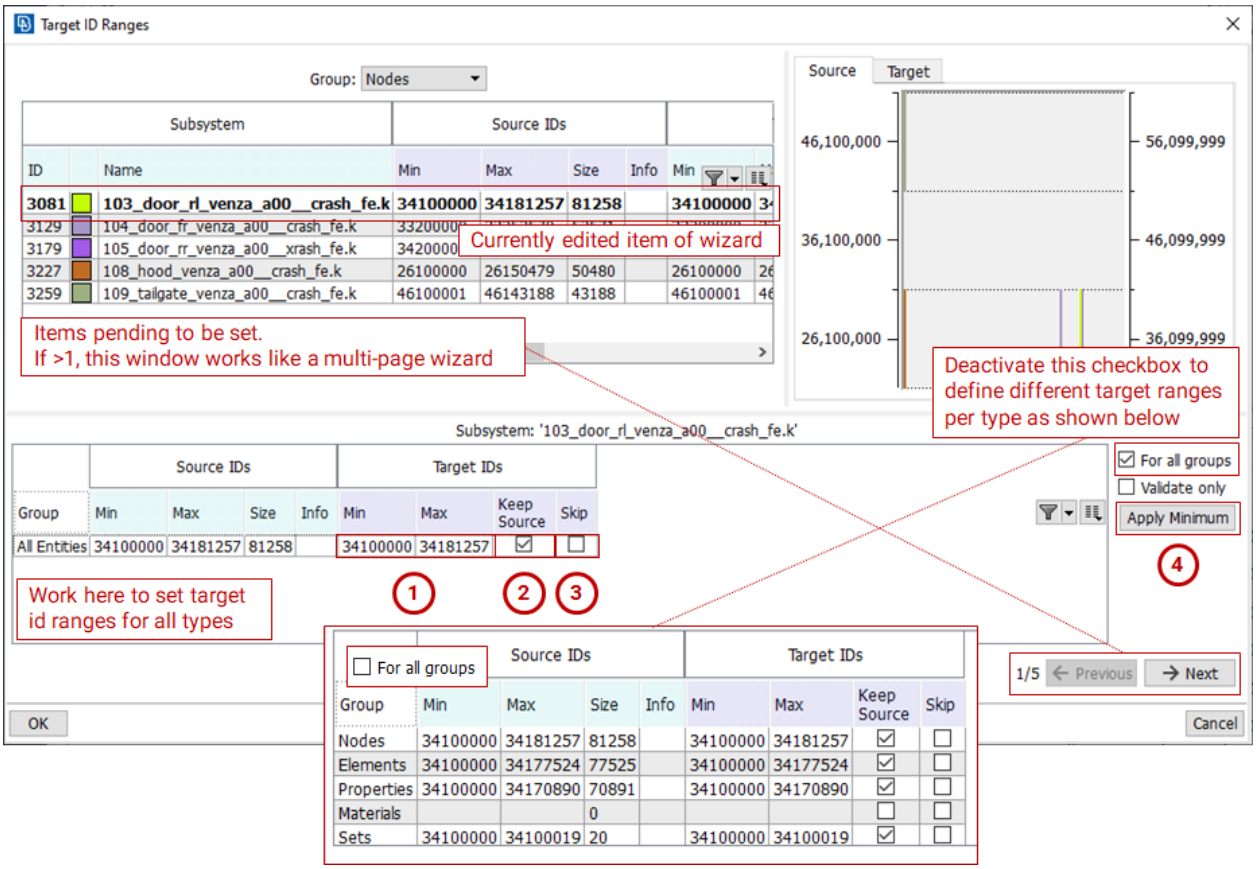
More information on these four processes is given in the following paragraphs.

5.2.1. Adaptation for the control of ids

The term “control of ids” essentially refers to the possibility to describe in a comprehensive manner the target id ranges for all the different modules of a composition.

A prerequisite for the definition of the target ranges is for the adapted module to already have information on its *source id ranges*. The *source id ranges* of a module are captured automatically with the *indexing process* the moment the module is added in DM, either through *Save in DM* or with *Add in DM*. More information on the population of attributes that describe the source ids can be found in paragraph 3.2

In the *Model Browser*, the target ids of an adapter are defined through the context menu option **Actions > Target Id Ranges**.



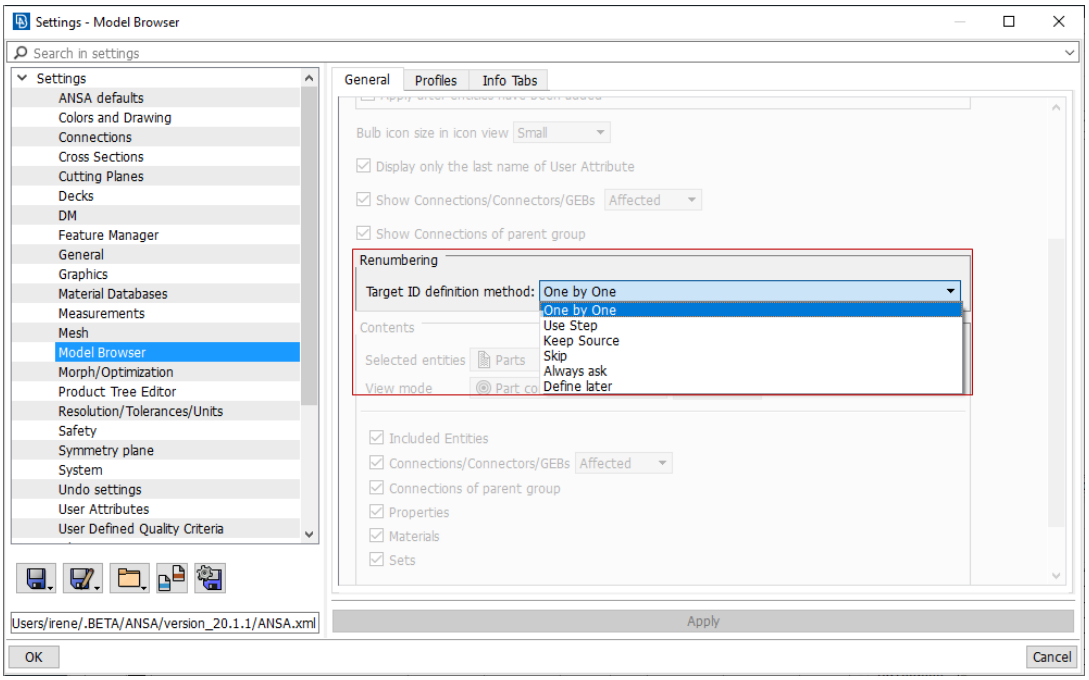
The *Target ID Ranges* window can work for a multi-selection of adapters. The top area lists the selected adapters. If more than one adapters are selected, the window behaves like a multi-page wizard and Next/Previous navigation buttons appear at the bottom right. In this case, each page edits the target ids of the adapter marked with bold fonts in the top list.

Target ranges are set in the bottom area. There are two modes supported: Setting a common target range for all entity types and setting a different target range for each different type. Switching between these modes is done by activating/deactivating the option “For all groups”.

There are 4 alternative methods for dealing with the adaptation of ids. The numbers correspond to the annotations in the image above:

1. Manually set target range min-max
2. Set the target range min-max automatically by copying the source range min-max (not recommended, as the target range will be an exact fit to the source, meaning that even the slightest modification in the number of entities of the module will lead to adaptation failure)
3. Skip. Leaves target ranges blank in order for the user to set them later.
4. Try to identify the minimum available range of adequate width based on existing targets

By default, when a base module is added to a compound Model Browser Container, the *Target ID Ranges* window pops-up. It is possible to change this behavior in the Model Browser Settings as shown below:

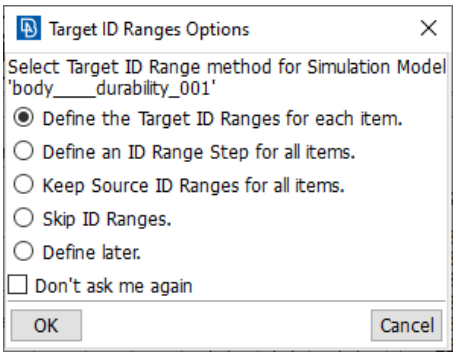


One by One *(default)*: *Target ID Ranges* window will pop-up every time a module is added to a compound Model Browser Container

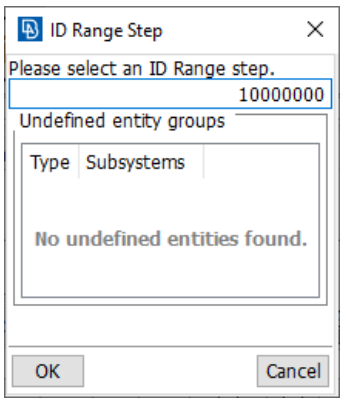
Keep Source: Option 2 described above

Skip: Option 3 described above

Define later: Do nothing. Target id ranges will be left blank. Identical to **Skip**.

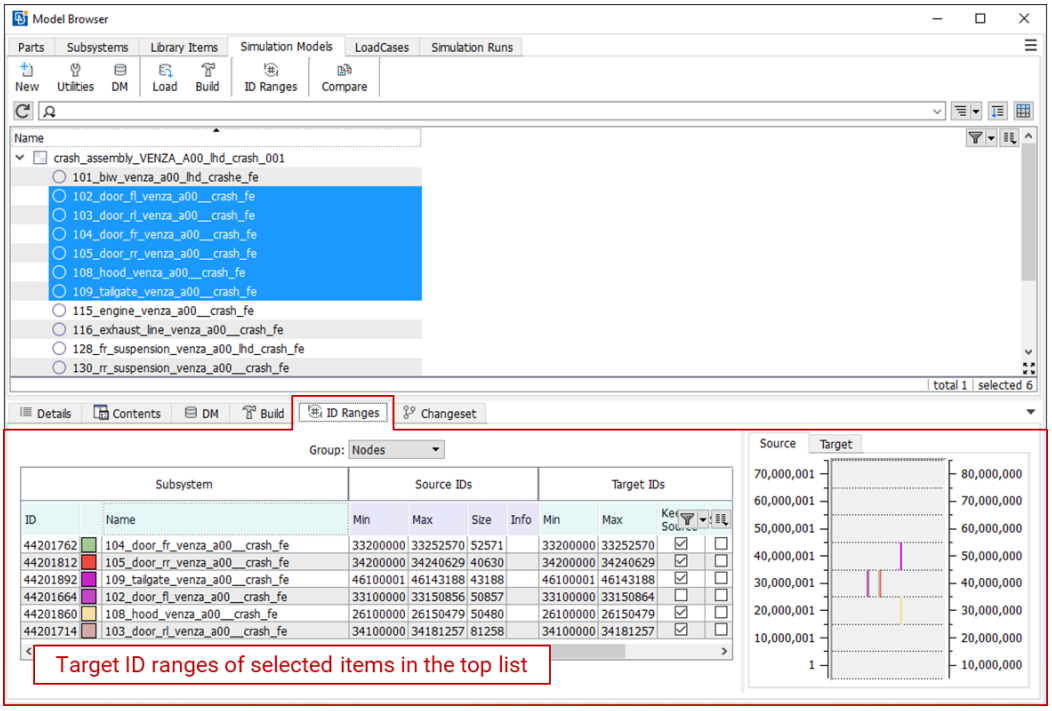


Always ask: A window pops-up that offers all alternative possibilities in a group of radio buttons



Use Step: This option only makes sense when a multi-selection of modules are added to a compound container. It allows the definition of a step value for the auto-generation of targets, starting from the minimum ids.

Alternatively, one can set the target id ranges with the aid of the *ID Ranges* bottom tab.



There are two main modelling practices supported in the Modular Environment:

Method 1: Renumber the modules to their target id range at the end of the module preparation phase, while still working on a modular basis. Then, during adaptation, only confirm that all modules indeed satisfy the prescribed target id ranges.

Method 2: Renumber all modules to a common target id range starting from *1*. Then, during adaptation, transform the ids. The transformation of ids can be achieved either with renumber of the module or with the use of the id transformation keywords that are supported by LS-Dyna and Radioss.

Although method 1 is the most widely used, there are cases where it must be combined with method 2. For example, in cases where the same Subsystem needs to be used more than once in the same assembly in different positions each time (i.e. in different instances), one of the instances will use the source ids (method 1) but all other instances will need to use a transformed id space (method 2). The same practice would also be applied in case a Library Item, e.g. a crash dummy, needs to be used more than once in the same loadcase in different position each time.

The adaptation of the ids for both methods is implemented in the Build Process of the adapters. The operation of the related Build Actions is affected by the Build Attribute **Id Handling Rule**. The tables below describe the related Build Actions and their behavior for each different value of this attribute:

Id Handling Rule	Validate Only
Solver	All
Build Action	ID Handling
It checks whether the source ids of the modules satisfy the target id specifications given in the adapter. If the source ids are compatible with the target ids, the Build Action gets OK status. In case there's the target range is not satisfied, or the source or target ids are not defined, the Build Action fails.	

Id Handling Rule	Renumber
Solver	Nastran, Abaqus, Pam-Crash
Build Action	ID Handling
It checks whether the source and target ids of are well defined. In case the source or target ids are not defined, the Build Action fails. Note: For Pam-Crash, in case the setting *enable_modules* which activates the working mode with Pam-Crash Modules, this Build Action does nothing.	
Build Action	Create Renumbered Deck File
If the source ids are compatible with the target ids, the Build Action does nothing and gets OK status. In case the target range is not satisfied, the source module is read in the background, it is renumbered with the specified target id ranges, and is saved in a temporary location. During the *Save in DM* of the parent Model Browser Container, the temporary file will be uploaded to DM as an adapted module, it will be stored under the parent Model Browser Container and its path will be included in the main file of the parent Model Browser Container. The Build Action may fail in case the source module does not exist in DM. Note: For Pam-Crash, in case the setting *enable_modules* which activates the working mode with Pam-Crash Modules, this Build Action does nothing.	

Id Handling Rule	Renumber
Solver	LS-DYNA
Build Action	ID Handling
It checks whether the source ids of the modules satisfy the target id specifications given in the adapter. If the source ids are compatible with the target ids, the Build Action does nothing and gets OK status. In case there's any deviation, it calculates the offset values for the 8 categories of entities supported in the *INCLUDE_TRANSFORM keyword as: Offset value = target min id - source min id Once the offsets have been calculated they will be used during the Output ot the parent Model Browser Container, in order to reference the base module with a properly configured *INCLUDE_TRANSFORM keyword	

Id Handling Rule	Renumber
Solver	Radioss
Build Action	ID Handling
It checks whether the source and target ids of are well defined. In case the source or target ids are not defined, the Build Action fails.	
Build Action	Create Renumbered Deck File
It checks whether the source ids of the modules satisfy the target id specifications given in the adapter. If the source ids are compatible with the target ids, the Build Action does nothing and gets OK status. In case there's any deviation, it calculates the offset values for the 6 categories of entities supported in the //SUBMODEL keyword as: Offset value = target min id - source min id Once the offsets have been calculated, it creates a temporary Radioss file in the background, containing a //SUBMODEL keyword with the proper offsets, using as *#include* the source path of the module.	

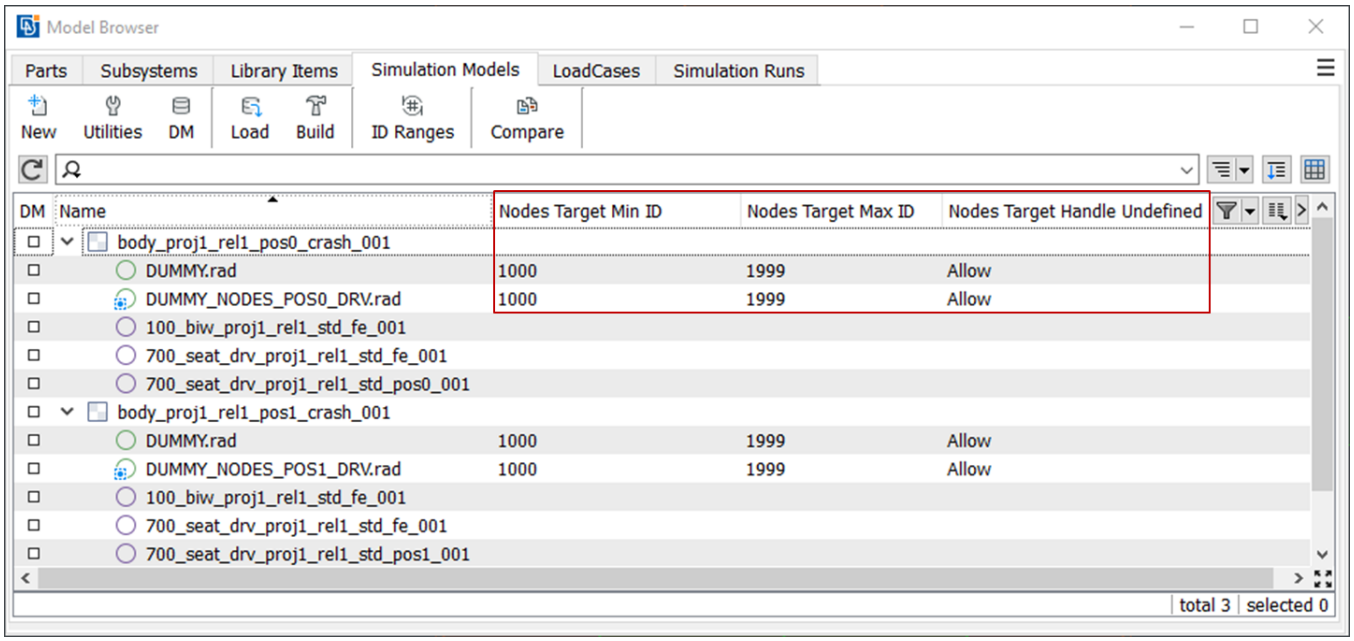
```
#RADIOSS STARTER
/BEGIN

      2019
//SUBMODEL/2
701_DOOR_STRIKER_FT_LT_STD_CRASH_FE_v001
         0       100       100       100         0         0
#---1----|----2----|----3----|----4----|----5----|----6----|----7----|----8----|----9----|--
-10----|
#include /disk1/DM/Subsystems/PRJ1/REL1/701_DOOR_STIKER/-
/CRASH/001/FE/Radioss/repr/701_DOOR_STRIKER_FT_LT_STD_CRASH_FE_v001.rad
//ENDSUB
/END
```

During the *Save in DM* of the parent Model Browser Container, the temporary file will be uploaded to DM as an adapted module, it will be stored under the parent Model Browser Container and its path will be included in the main file of the parent Model Browser Container.

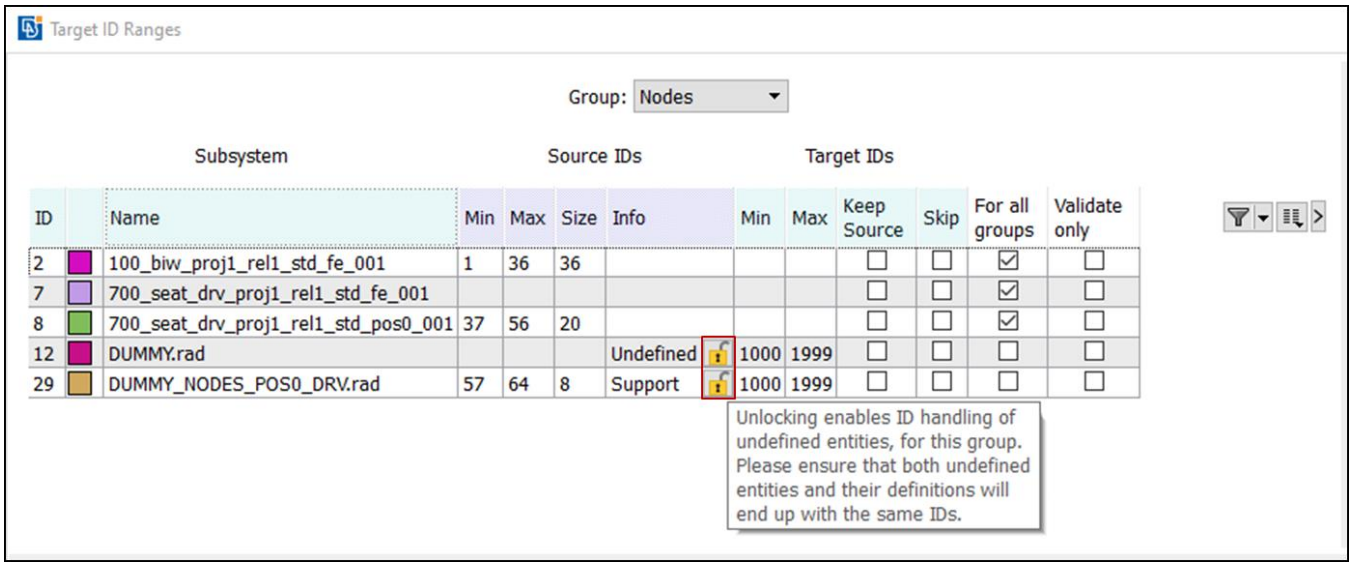
5.2.1.1. Handling of complementary includes

It is not uncommon for specific systems of a model to have their nodes separated from the rest of the entities and handled as a separate include file. Characteristic examples are the seat and dummy models that often have their nodes separated in a different include for each different position required by the loadcases. The handling of such systems, that consist of more than one complementary includes, poses some challenge in the compilation of the Simulation Run main files, as the target id ranges of all complementary includes must be identical.

In the Modular Environment, such complementary includes can be managed as different Subsystems or Library Items, allowing the definition of identical target id ranges for the entity types like nodes, that are defined in one Subsystem/Library Item and are only referenced in the other.



In the example above, two Simulation Models include the same dummy as a driver, once in position 0 and once in position 1. In these two instances, the dummy nodes and the rest of the dummy entities are split in different Library Items. When used in the same Simulation Model, the two Library Items can get identical target id ranges, using the option **Nodes Target Handle Undefined** activated through the *Target ID Ranges* tool, to enable the proper id management.



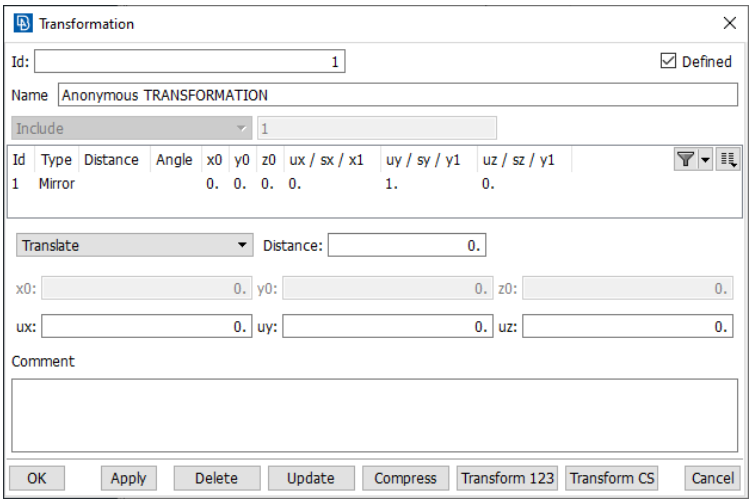
5.2.2. Adaptation for the control of positions

The term “control of positions” essentially refers to the possibility of using a module in a position different from the nominal position where it was originally designed. The Modular Environment supports two different methodologies for the control of the final position of a module:

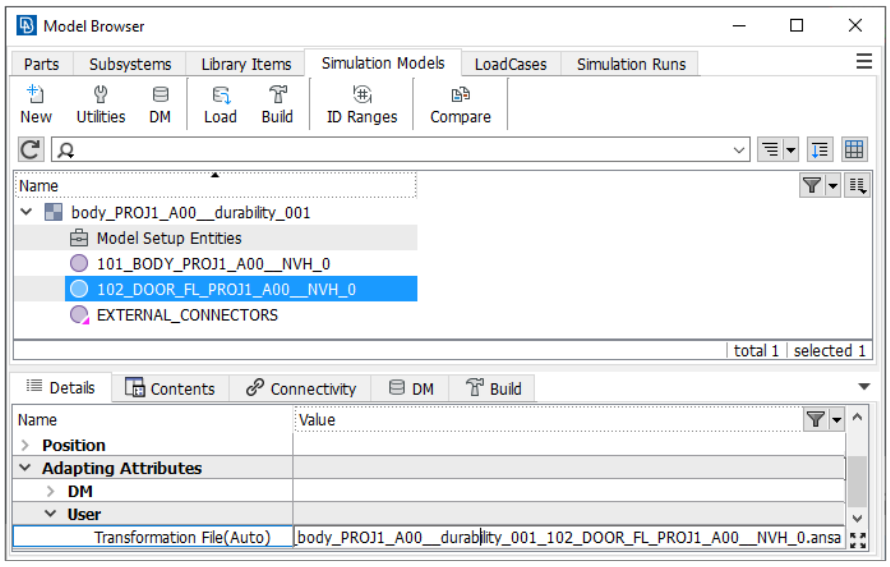
Method 1: Through the definition of a transformation card. A transformation card may encapsulate a series of geometric transformations of types *translate*, *rotate, scale, mirror* that are applied to *all the nodes* of the module. This transformation method is applicable in all cases where all the nodes of the module must be positioned with a single transformation, e.g. for the positioning of a barrier, the re-use of a front left door on the right side, etc.. In these cases, the transformation of the adapter is defined through the context menu option *Actions > Define Transformation* and the application of the adaptation is carried-out by the Build Action *Transformations*.

Method 2: Through the definition of a kinetic position. In cases where the module contains a kinetic mechanism, e.g. a convertible roof-top, a seat or a crash dummy, positioning the module requires the application of a different transformation to each member of the mechanism. In these cases, the transformation of the adapter is defined through the context menu option *Actions > Positioning* and the application of the adaptation is carried-out by the Build Action *Positioning*.

Method 1: Positioning through transformation cards

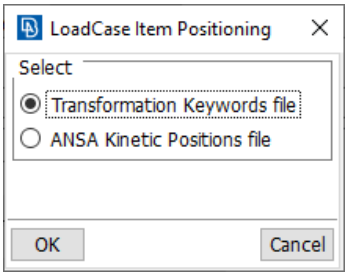


Using the function **Actions > Define Transformation** on a Simulation Model or Loadcase Adapter, the transformation card of the current deck pops-up in order for the user to define the proper transformation components.



On OK, the transformation card is written in a temporary file and its automatically associated with the adapter through the adapting attribute **Transformation File(Auto)**

Note that if the module is loaded, pressing OK in the transformation card will actually apply the transformation on the loaded module, moving it accordingly.



In case the transformation card is already available in the form of a solver keyword file, it is possible to associate it with the adapter directly through *Actions > Positioning,* by selecting the option “Transformation Keyword File”.

In this case, the transformation file is kept under the adapter attribute **Transformation File**

Once either of the attributes is set, the Build Action *Transformations* can be used to generate the required adaptation metafiles. Depending on the solver, the products of this process are different. More information is given in the tables below:

Solver	Nastran (does not support transformation keyword)
Build Action	Transformations
It checks whether the transformation file defined under “Transformation File(Auto)” attribute is valid, loads the source module in the background, applies the transformation and saves it in a temporary location. During the *Save in DM* of the parent Model Browser Container, the temporary file will be uploaded to DM as an adapted module, it will be stored under the parent Model Browser Container and its path will be included in the main file of the parent Model Browser Container.	

Solver	Abaqus
Build Action	Transformations
It checks whether the transformation file defined under Transformation File(Auto) attribute is valid, loads the source module in the background and creates a node set with all the nodes of the module. Finally, during the *Save in DM* of the parent Model Browser Container, the *NSET is written in the main file of the parent Model Browser Container, together with the *NMAP that corresponds to the transformation defined.	

Solver	Pam-Crash
Build Action	Transformations
It checks whether the transformation file defined under Transformation File(Auto) attribute is valid, loads the source module in the background and creates a group with all the nodes of the module. Finally, during the *Save in DM* of the parent Model Browser Container, the GROUP/ is written in the main file of the parent Model Browser Container, together with the TRSFM / that corresponds to the transformation defined.	

Solver	LS-Dyna
Build Action	Transformations
It checks whether the transformation file defined under Transformation File(Auto) attribute is valid. During the *Save in DM* of the parent Model Browser Container, the transformation is written in the main file of the parent Model Browser Container under a *DEFINE_TRANSFORMATION keyword whose id is finally used under the *INCLUDE_TRANSFORM that is used in order to include the transformed module.	

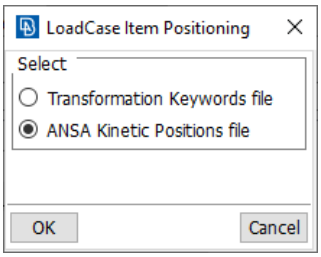
Solver	Radioss
Build Action	Transformations

It checks whether the transformation file defined under “Transformation File(Auto)” attribute is valid and creates a temporary Radioss file in the background, containing a //SUBMODEL keyword using as *#include* the source path of the module and a /TRANSFORM keyword, usi
During the *Save in DM* of the parent Model Browser Container, the temporary file will be uploaded to DM as an adapted module, it will be stored under the parent Model Browser Container and its path will be included in the main file of the parent Model Browser Container

```
#RADIOSS STARTER
/BEGIN
auto_created_transf_abs_102_DOOR_FL_VENZA_A00_0
      2019
//SUBMODEL/5000064
102_DOOR_FL_VENZA_A00_0
         0         0        -2         0         0         0
#---1----|----2----|----3----|----4----|----5----|----6----|----7----|----8----|----9----|--
-10----|
#include /disk1/DM/Subsystems/102_DOOR_FL/RIGID_ELEMENTS/VENZA/A00/001/-
/Radioss/common/repr/102_DOOR_FL_VENZA_A00_0.rad
//ENDSUB
#---1----|----2----|----3----|----4----|----5----|----6----|----7----|----8----|----9----|--
-10----|
/TRANSFORM/ROT/2000002
Anonymous TRANSFORM
            1051.0045023007613  -897.9605557134859   698.8988140146448
5000064
                     1052.4321           -886.6464            748.6132               180.
/END
```

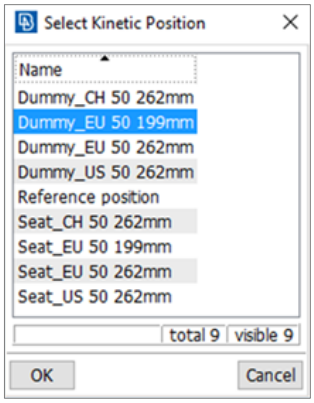
During the *Save in DM* of the parent Model Browser Container, the temporary file will be uploaded to DM as an adapted module, it will be stored under the parent Model Browser Container and its path will be included in the main file of the parent Model Browser Container.

Method 2: Positioning through kinetic positions



Using the function *Actions* > *Positioning* on a Simulation Model or Loadcase Adapter, the user is prompted to specify a file that contains KINETIC_POSITIONs that have been built on the same module.

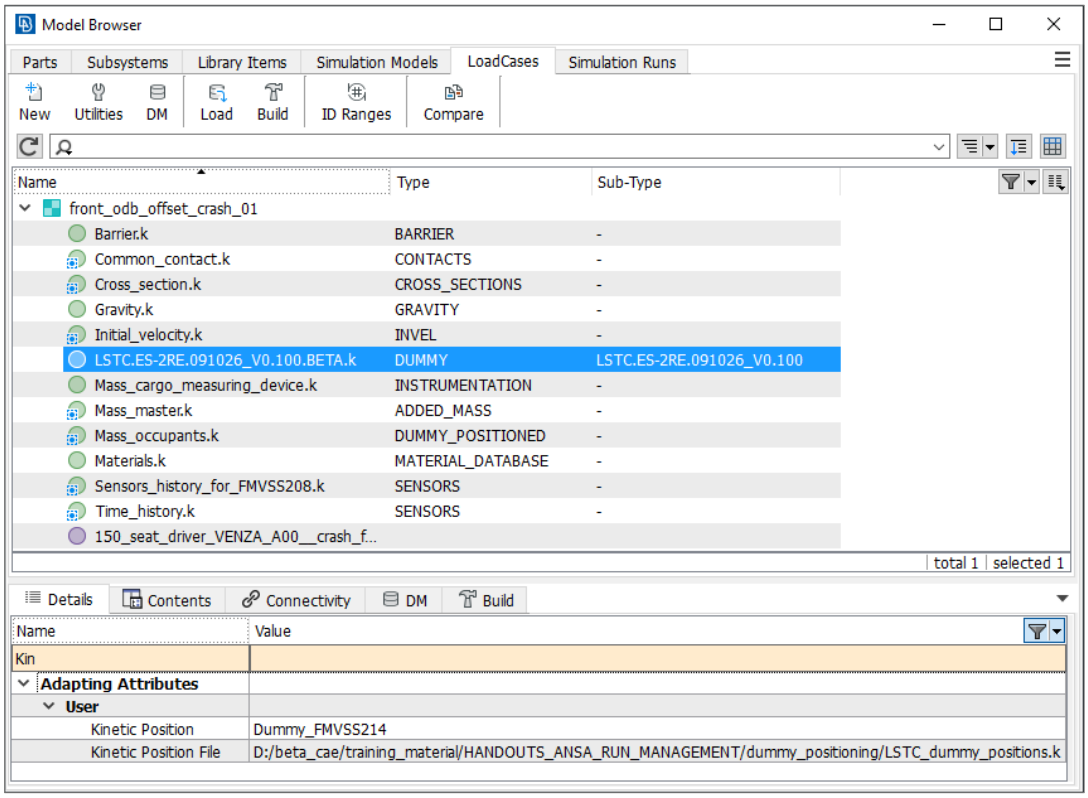
Select the option **ANSA Kinetic Positions file** to either select an ANSA file containing kinetic positions or a solver keyword files where the kinetic positions are written as ANSA comments. On **OK** the user is prompted to specify the Kinetic Positions file.



Once the file has been located through the file manager, it is scanned and a list with all available kinetic positions pops-up.

In this window the user needs to select a kinetic position by name.

Finally, the kinetic positions file and the name of the kinetic position are stored under the adapter attributes **Kinetic Position File** and **Kinetic Position.**



Once the adapting attributes are defined, the Build Action *Positioning* can be used to generate the required adaptation metafiles. Depending on solver, the products of this process are different. More information is given in the tables below:

Solver	Nastran (does not support transformation keyword)
Build Action	Positioning
It checks whether the **Kinetic Position File** and the **Kinetic Position** defined are valid, loads the source module and the kinetic position file in the background, applies the kinetic position and saves the transformed module in a temporary location. During the *Save in DM* of the parent Model Browser Container, the temporary file will be uploaded to DM as an adapted module, it will be stored under the parent Model Browser Container and its path will be included in the main file of the parent Model Browser Container.	

Solver	Abaqus
Build Action	Positioning
It checks whether the **Kinetic Position File** and the **Kinetic Position** defined are valid, loads the source module and the kinetic position file in the background, applies the kinetic position and saves in a temporary location an adaptation metafile (*internal transformation file*) containing the definitions of the node-sets being transformed (*NSET) and the transformation keywords (*NMAP) for each and another metafile (*external transformation file*) containing two include keywords, one to include the adapting metafile with the transformations and one to include the source module. During the *Save in DM* of the parent Model Browser Container, both temporary files will be uploaded to DM and will be stored under the parent Model Browser Container. Finally, the *external transformation file* will be included in the main file of the parent Model Browser Container.	

Solver	Pam-Crash
Build Action	Positioning
It checks whether the **Kinetic Position File** and the **Kinetic Position** defined are valid, loads the source module and the kinetic position file in the background, applies the kinetic position and saves in a temporary location an adaptation metafile (*internal transformation file*) containing the definitions of the node-sets being transformed (GROUP /) and the transformation keywords (TRSFM /) for each and another metafile (*external transformation file*) containing two include keywords, one to include the adapting metafile with the transformations and one to include the source module. During the *Save in DM* of the parent Model Browser Container, both temporary files will be uploaded to DM and will be stored under the parent Model Browser Container. Finally, the *external transformation file* will be included in the main file of the parent Model Browser Container.	

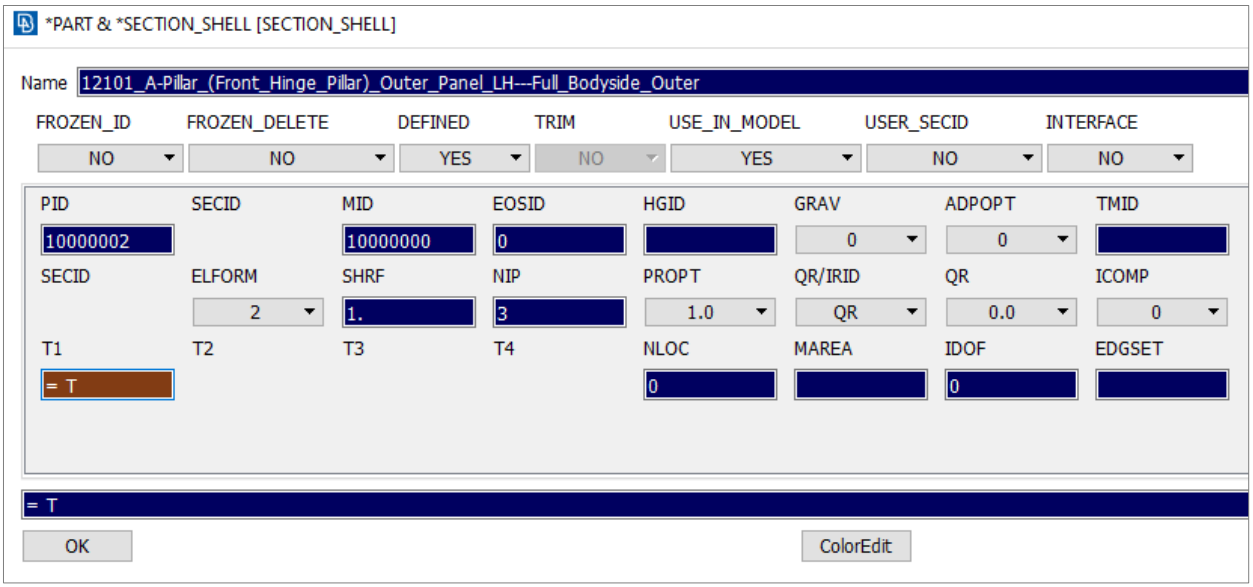
Solver	LS-Dyna
Build Action	Positioning
It checks whether the **Kinetic Position File** and the **Kinetic Position** defined are valid, loads the source module and the kinetic position file in the background, applies the kinetic position and saves in a temporary location an adaptation metafile (internal transformation file) containing the definitions of the node-sets being transformed (*SET_NODE_LIST) and the transformation keywords (*NODE_TRANSFORM) for each and another metafile (external transformation file) containing 2 include keywords, one to include the adapting metafile with the transformations and one to include the source module. During the Save in DM of the parent Model Browser Container, both temporary files will be uploaded to DM and will be stored under the parent Model Browser Container. Finally, the external transformation file will be included in the main file of the parent Model Browser Container.	

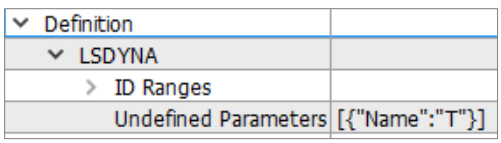
Solver	Radioss
Build Action	Positioning
It checks whether the **Kinetic Position File** and the **Kinetic Position** defined are valid, loads the source module and the kinetic position file in the background, applies the kinetic position and saves in a temporary location an adaptation metafile (internal transformation file) containing the definitions of the node-sets being transformed (/GRNOD/GENE) and the transformation keywords (/TRANSFORM) for each and another metafile (external transformation file) containing 2 include keywords, one to include the adapting metafile with the transformations and one to include the source module. During the Save in DM of the parent Model Browser Container, both temporary files will be uploaded to DM and will be stored under the parent Model Browser Container. Finally, the external transformation file will be included in the main file of the parent Model Browser Container.	

5.2.3. Adaptation for the control of parameters

In the Modular Run Environment it is possible to exploit the parameterization capabilities of the solvers in order to use parametrically defined modules. The main idea can be described through the example of a Subsystem that contains a parametrically defined thickness value in one of its shell properties. The property card of the Subsystem makes reference to an “undefined” parameter with name, say, *T*. The requirement is to be able to modify the value of *T* without opening and editing the Subsystem file.

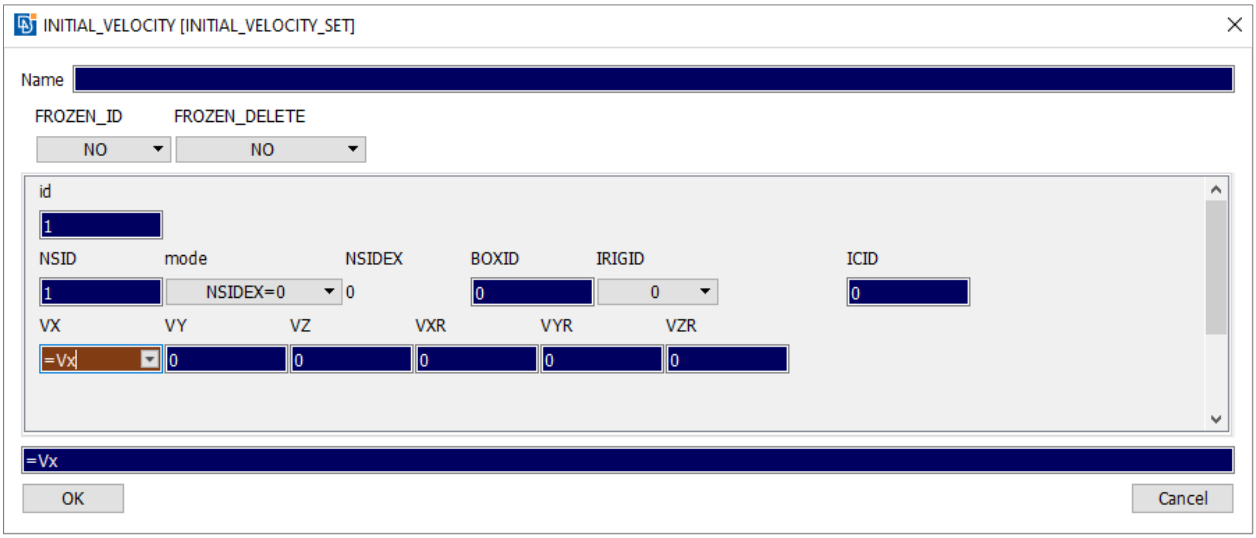
In the example below, the Subsystem contains a parametrically defined thickness for one of its properties.



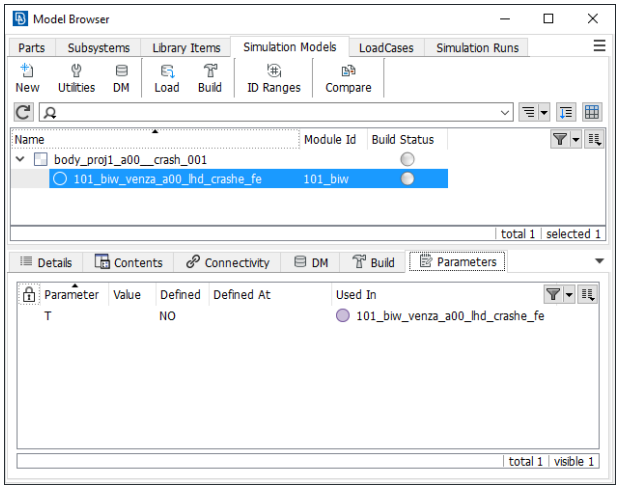


Saving this Subsystem in DM (or adding it in DM through the DM Browser), the indexing process will extract information on the parameters referenced in the file (both defined and undefined parameters).

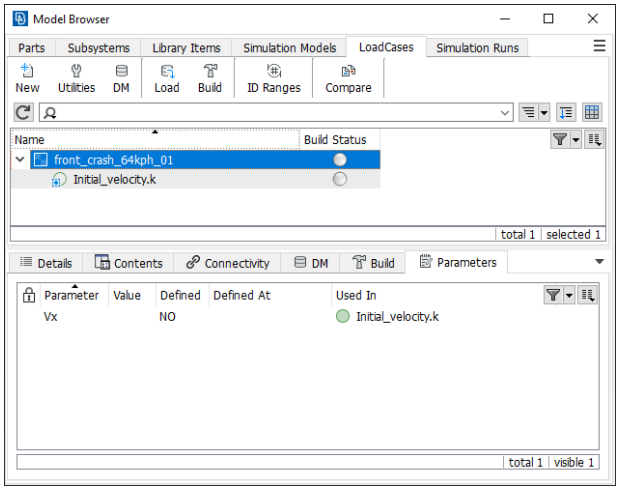
At the same time, a loadcase file that contains an initial velocity card contains a parametric definition of the velocity magnitude.



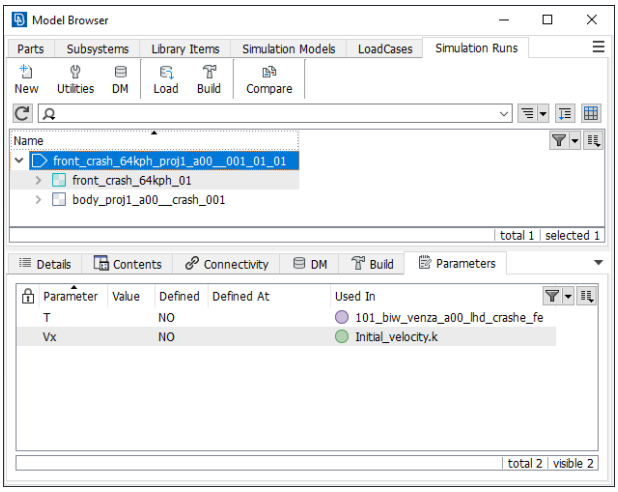
In order to be able to control the value of the parameters *T* and *Vx* without modifying the source modules, the parameters must be defined on the adapter level or on any higher level container. In the Modular Environment, parameters are controlled in the **Parameters** bottom tab of the Model Browser:



Selecting the Simulation Model Adapter of a parametric Subsystem, all its parameters are listed in the Parameters tab.

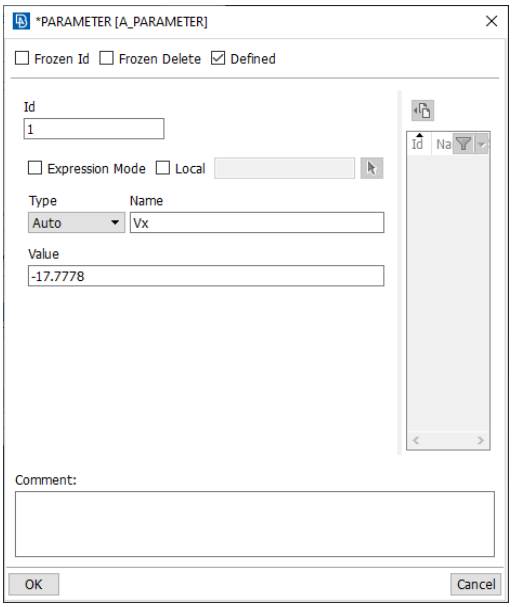


Selecting any higher level container, all contained parameters are listed in the Parameters tab. In the screenshot on the left, the Loadcase is selected, that contains a parametrically defined Library Item.

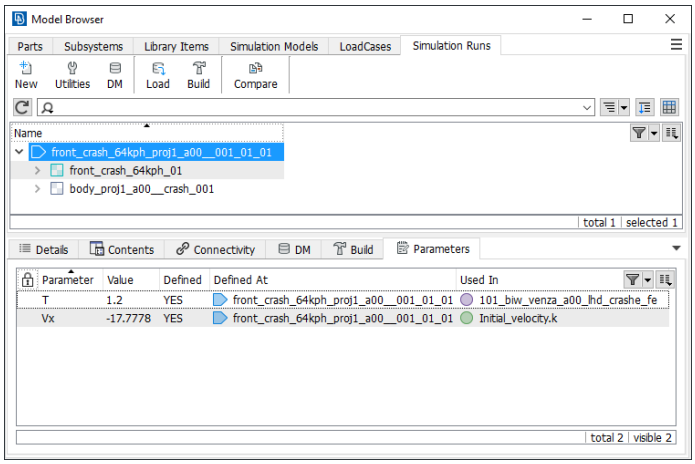


In the screenshot on the left, the Run is selected, that contains a Simulation Model with the parametric Subsystem and the Loadcase with the parametric Library Item. In this case, all related parameters are listed.

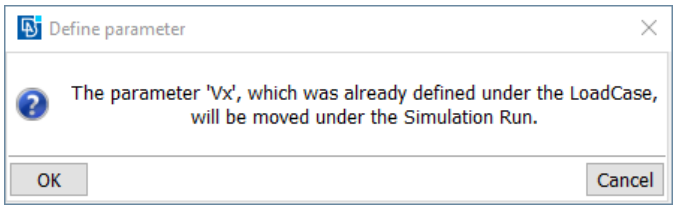
To define the parameters, the user needs to double-click on each row of the Parameters List.



Double-click opens the card of A_PARAMETER in order for the user to define the value of the parameter.



Note that the parameter will be finally defined in the compound Model Browser Container that was selected the moment the parameter value was defined. The “owner” of the parameter is shown in the “Defined At” column of the Parameters List. The owner is the entity in which the parameter keyword will be finally written.



In cases where the parameter was already defined under a different container and the editing of its value will change the container where it is defined, as described above, a confirmation message pops-up to enable the modification of the parameter value only, without changing its container.

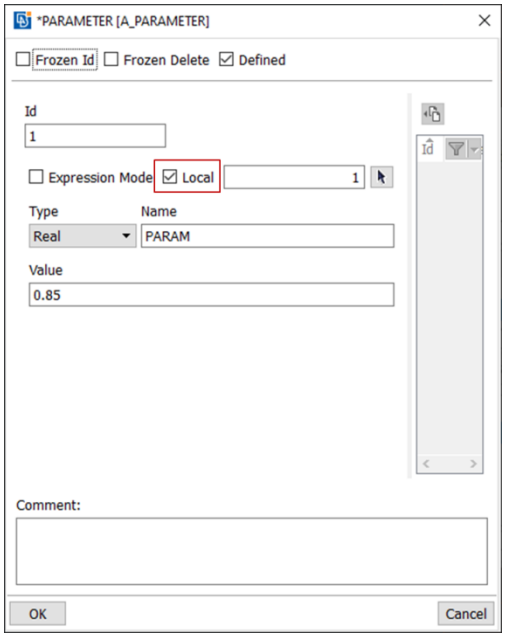
Pressing “OK” in the message window shown above, will move the parameter definition to the container that was selected when its value was modified (e.g. the Simulation Run), while pressing “Cancel” will result in the parameter definition to remain in the container where it was initially defined.

Writing out the Simulation Run selected above, where the parameters T and Vx have been initialized, leads to the following text in the keyword file of the Simulation Run.

```
$============================DM INFO BEGIN============================
$ANSA_DM_INFO_PRE_FORMAT;
$
$Entity Type      : Simulation Run
$Iteration        : 01
$Simulation_Model : 25
$LoadCase         : 26
$File Type        : LsDyna
$File             :
$Name             : front_crash_64kph____001_01_01
$User             : user1
$Solver Version   : R11.1
$
$END_ANSA_DM_INFO_PRE_FORMAT;
$============================DM INFO END==============================
$
*KEYWORD
*PARAMETER
R Vx        -17.7778
R T              1.2
…
```

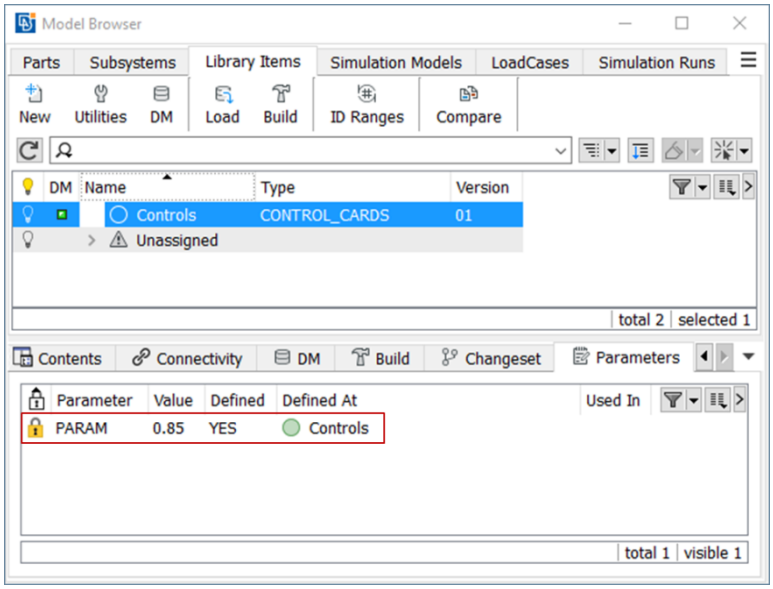
5.2.3.1. Handling of local parameters

When the target solver is LS-Dyna, where the parametrization capabilities include the definition of local parameters, ANSA supports the handling of parameters on base module level. A characteristic example is the case of a Library Item that contains a parametric definition, which is defined locally and should not be modified when the Library Item is used in a compound Model Browser Container.



The definition of a parameter as local is done through the respective option in the A_PARAMETER Edit Card. Writing out the Library Item with the parameter defined as local leads to the following text in the keyword file of the Library Item.

```
$================DM INFO BEGIN==================
$ANSA_DM_INFO_PRE_FORMAT;
$
$Entity Type  : Library Item
$Type         : CONTROL_CARDS
$Sub-Type     : -
$Version      : 01
$Name         : Controls
$File         :
$Library Type : Library_File
$User         : user1
$
$END_ANSA_DM_INFO_PRE_FORMAT;
$================DM INFO END====================
$
*KEYWORD
*PARAMETER_LOCAL
R PARAM         0.85
…
```



When the definition of a base module containing local parameters is loaded from DM, the parameter definition is locked and cannot be modified, even when the base module is added to a compound Model Browser Container.

Only when the contents of the base module are loaded can the value of the parameter be modified, in which case it will be marked as modified (with a magenta DM Update Status).

5.2.4. Adapting attributes for the control of the structure of the main files

Adapting Attributes	
DM	
IO Index	3
Output Option	Standard include output

There are two built-in adapting attributes that are used in order to control the structure of the solver keyword file of a compound Model Browser Container.

- **IO Index**: This attribute is an integer that represents the order in which this child container will appear within the solver keyword file of its parent compound Model Browser Container.

Name	IO Index
crash_assembly_VENZA_A00_lhd_crash_001	
101_biw_venza_a00_lhd_crashe_fe.k	1
102_door_fl_venza_a00__crash_fe.k	2
103_door_rl_venza_a00__crash_fe.k	3
104_door_fr_venza_a00__crash_fe.k	4
105_door_rr_venza_a00__crash_fe.k	5
108_hood_venza_a00__crash_fe.k	6
109_tailgate_venza_a00__crash_fe.k	7
115_engine_venza_a00__crash_fe.k	8
116_exhaust_line_venza_a00__crash_fe.k	9
128_fr_suspension_venza_a00_lhd_crash_fe.k	10
130_rr_suspension_venza_a00__crash_fe.k	11
133_fuel_tank_venza_a00__crash_fe.k	12
150_seat_driver_venza_a00__crash_fe.k	13
154_radiator_venza_a00__crash_fe.k	14
Patches.k	15

In the screenshot on the left one can see the IO Index of all the Subsystems of a Simulation Model. It denotes that the first include keyword to be written in the keyword file of the Simulation Model is the 101_biw and the last is the Patches.

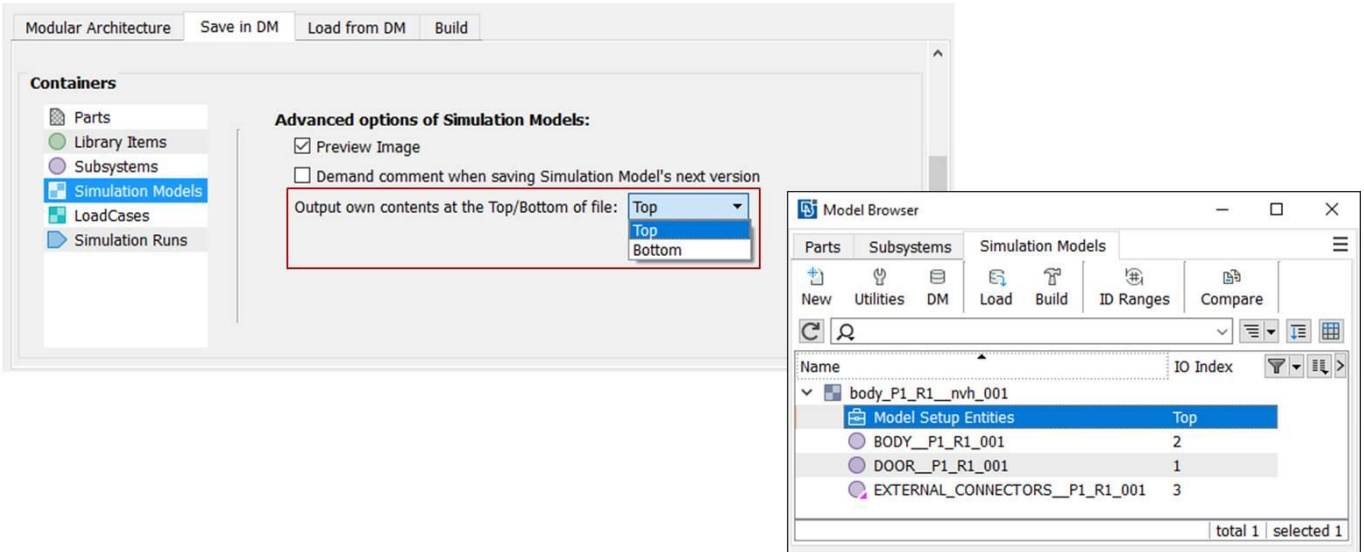
The IO Index can be edited manually. ANSA makes sure that the same IO Index will never be assigned to more than one adapters of the same compound Model Browser Container.

- **Output Option**: This attribute is used to describe the format in which a child container will be written within the solver keyword file of its parent compound Model Browser Container. It can get 4 alternative values:
  - **Standard include output**: When the parent container is saved as a keyword file, this item will be saved in DM as a monolithic include and an include keyword reference will be added in the solver keyword file of the parent container (equivalent to include output options: read-only = no / inline = no)
  - **Write content in main**: When the parent container is saved as a keyword file, this item will be written directly in the solver keyword file of the parent container (equivalent to include output options: read-only = no / inline = yes)
  - **Write keyword in main**: When the parent container is saved as a keyword file, this item will only be referenced with an include keyword in the solver keyword file of the parent container. This is the standard behaviour in all cases the parent container is saved with the option “references” (equivalent to include output options: read-only = yes / inline = no)
  - **Copy content in main**: When the parent container is saved as a keyword file, this item will be written directly in the solver keyword file of the parent container by copying it line-by-line from its source location in DM (equivalent to include output options: read-only = yes / inline = yes). This is the standard behaviour in all cases the parent container is saved with the option “monolithic”.

The reason why this setting is kept on the adapter level is in order to be able to select different Output Option for each of the contents of a compound Model Browser Container. For example, a Subsystem may need to be written out as an include keyword reference whereas some other may need to be written “inline” the main file.

In order to force these values during output, the option **Existing Setup** must be specified in the Output Options of the parent Model Browser container being saved.

Apart from base Modules, compound Model Containers may also contain Model Setup Entities or Connections of their own. To control whether these entities will be written at the top or at the bottom of the solver keyword file, a setting in the Modular Environment Profile can be used. More specifically, in the *Save in DM* tab of the **ANSA Settings > Modular Environment**, it is possible to define for each compound container whether its own contents should be written at the top or bottom of the file. The selected value is also displayed in the *DM/IO index* column in the Model Browser.



5.3. Adaptation of loaded entities

For the adaptation of ids and for the handling of parameters it is mandatory for the program to have access to the “source” attributes of each module. As seen in the paragraphs above, in case of ids adaptation, it is necessary to know which are the source id ranges within the Subsystem/Library Item include files. Similarly, for the adaptation of parameters, it is required to know which are the parameters referenced within each module, either defined or undefined.

Every time a module is loaded in ANSA, either as a whole file or as an empty placeholder, it brings along its *Definition* metadata, which is the set of indexing metadata of the DM Object.

In cases where the base modules that require adaptation are fully loaded in ANSA, the retrieval of information about the “source” becomes ambiguous: Where should the program retrieve this information from? From the attributes of the corresponding Model Browser Container or from the currently loaded entities?

To resolve this ambiguity, a simple rule is followed, based on the *DM Update Status* of the adapted Model Browser Container:

- *DM Update Status* = green: Use the *Definition Attributes* of the Model Browser Container. The fact that the status is green means that the Model Browser Container has not been modified since loading from DM. Therefore, the *Definition Attributes* that were passed to the Model Browser Container the moment it was created are still reliable and are the ones to be used for best performance.
- *DM Update Status* = magenta: Re-index the Model Browser Container on the fly, update the *Definition Attributes* and use the updated values. The fact that the status is magenta means that the Model Browser Container has been modified since loading from DM. Therefore, the *Definition Attributes* that were passed to the Model Browser Container the moment it was created are unreliable and should not be used for adaptation. In this case, the required information will be extracted from the loaded entities and the *Definition Attributes* of the Model Browser Container will be updated.

The only exception to this rule is for base modules that are marked as **Read Only**: The Definition Attributes of Read Only Subsystems and Library Items are always retrieved from DM, even if their *DM Update Status* is magenta.

5.4. Un-adaptation

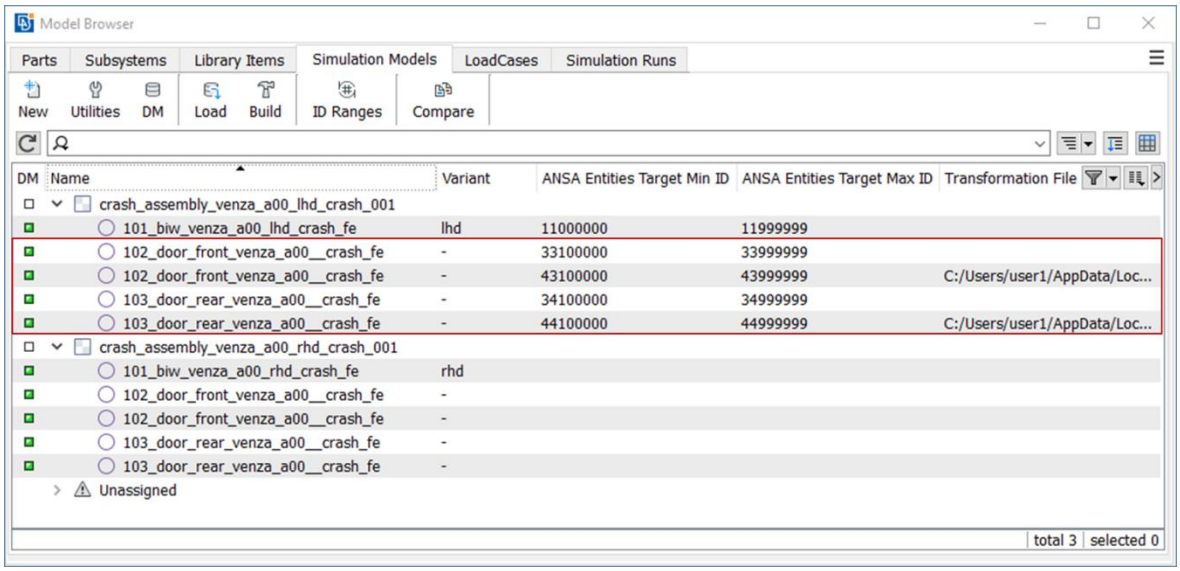
The term “un-adaptation” refers to the process of removing the adaptation characteristics from a Model Browser Container in order to bring it back to the state right before the adaptation. The possibility to “un-adapt” a Model Browser Container is very attractive, since it enables the saving of a *Definition File* of a base module from within an “assembly” model, where the module participates in an adapted state, together with other modules. There are 3 cases where un-adaptation would be desired:

1. Un-adaptation of ids: During save of a module that is already adapted by its parent container, any offsets could be removed in order to revert the ids to their source range. In v20.1.1 this un-adaptation is applied when saving the *definition* file of the module in ANSA format.
2. Un-adaptation of position: During save of a module that is already adapted by its parent container, any geometric transformations applied to the nodes of the module could be removed in order to revert the ids to their source range. This is not yet possible.
3. Un-adaptation of modifications made on interface points by connectors: During the realization of intermodular connectors of type B, as described in paragraph 6.1.2, the Assembly Points used to mark the interface points of each module are modified. During save of a module that takes part in such intermodular connection, the interface points could be restored to their original state. This un-adaptation is applied when saving the *definition* file of the module in ANSA format.

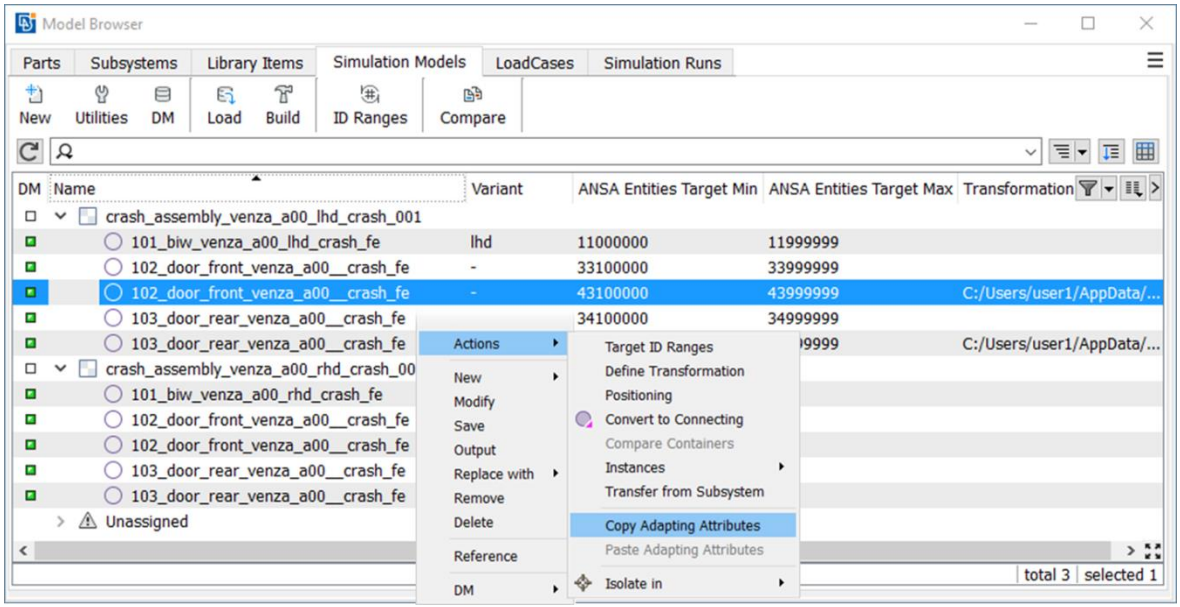
5.5. Synchronizing adaptation attributes between adapters

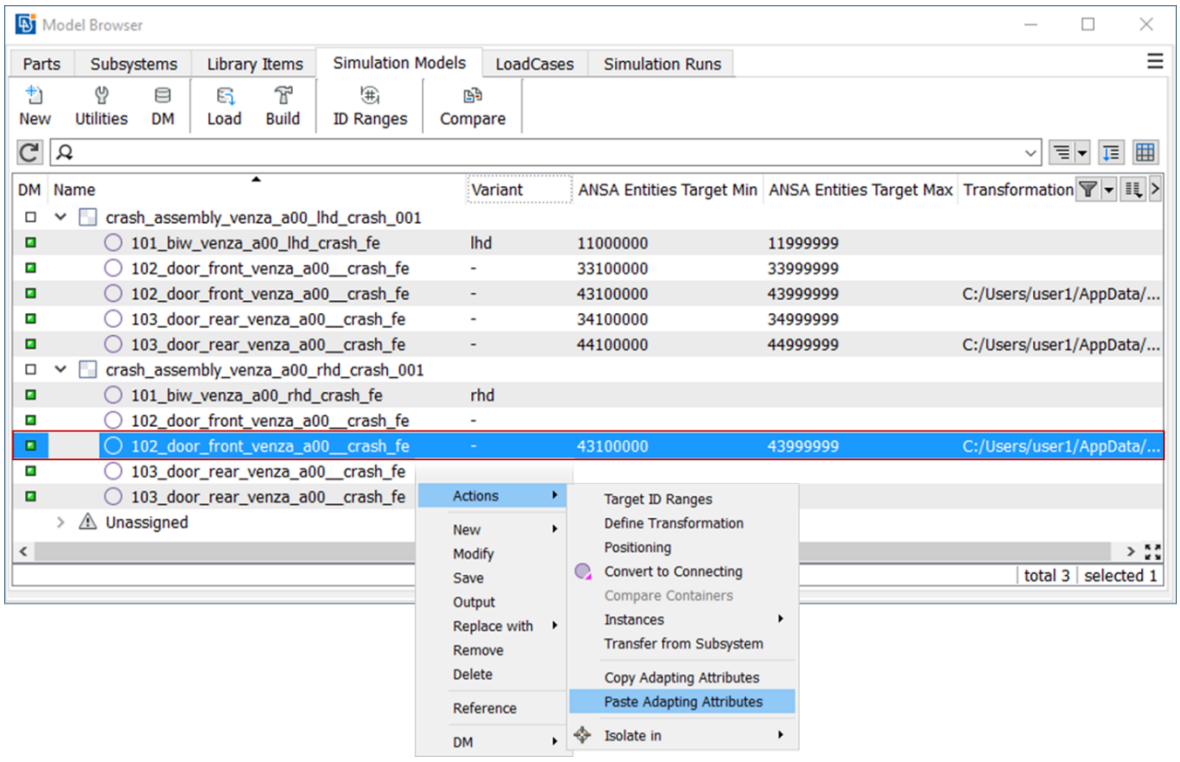
Working in a Modular Environment, it is quite often necessary to reuse the same target id ranges or transformations between the adapters of various Simulation Models or Loadcases. A characteristic example would be the use of several common Subsystems in two different Simulation Model variants. Since the same Subsystems need to be re-used, the transformation and target id ranges would have to be defined several times.

In the image below, two different Simulation Model variants have been defined, each containing two instances for each of the front and rear door Subsystems, since they would need to be re-used on both the left and the right hand side of the vehicle. The target id ranges for the door Subsystems and the transformations for one of the two instances of the front and rear door Subsystems have been defined for the first Simulation Model.

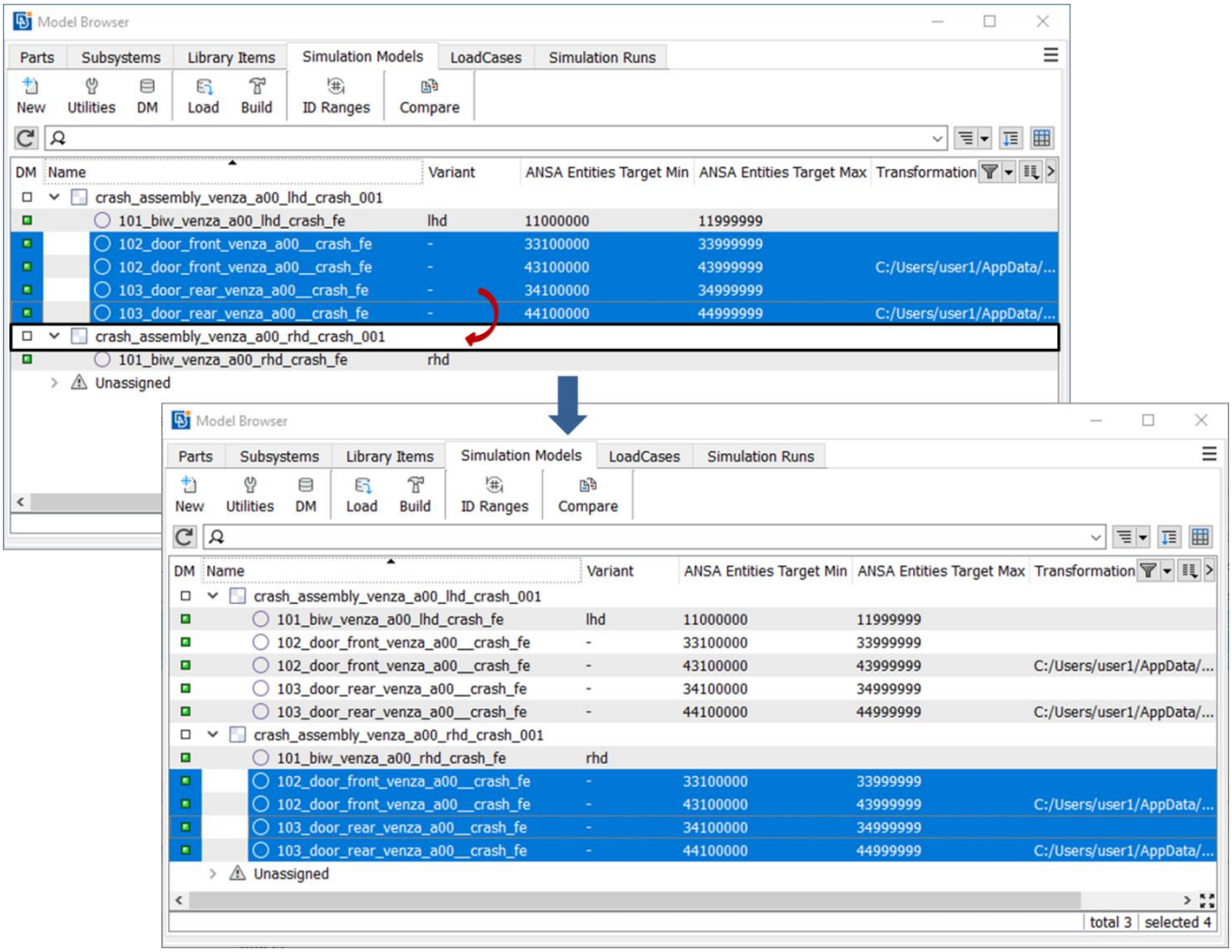


To facilitate the reuse of the same adaptation in the second model, so that the user does not have to define everything from scratch, ANSA enables the synchronization of the adapting attributes between the contained Model Browser Containers of the two Simulation Models, using the respective options **Actions > Copy Adapting Attributes** and **Actions > Paste Adapting Attributes** from the context menu of the adapters.





Alternatively, the synchronization of the adapting attributes between compound Model Browser Containers is facilitated with the use of drag-and-drop-operations.



In the example shown above, and assuming that the definition of the first Simulation Model variant exists and the user is in the process of creating the second model variant, selecting the desired adapters from the first Simulation Model variant and dropping them on the second Simulation Model, ANSA will add the selected Subsystems and copy their adapting attributes from the source model container to the target.

6. Model Assembly

Model Assembly in the Modular Run Environment is the process that manages the interconnection of different modules. There are three main types of connections that are included in the broad scope of Model Assembly:

1. Point connections: Managed by Connector Entities or pure FE entities
2. Continuous connections: Managed by Connection Lines/Faces
3. Set-based connections: Managed by Assembly Sets

There's a big variety of FE-representations for intermodular connections, usually depending on the solver used and the methods applied by each CAE team. Sometimes different flavors are limited to the use of different element types for the interface elements (e.g. RBE2s, RBE3s, RBODYs, MPCs, *COUPLINGs, rigid shells, etc.) and the body element (e.g. CELASx, CBUSHx, SPRINGs, BEAMs, RBE2s, *CONSTRAINED_RIGID_BODY etc.). However, there are several cases where more "exotic" FE-representations are employed, whose conception mainly satisfied the need for "automatic" assembly without the need of loading the FE representations of all the modules in the pre-processor. Examples of such connections are those that employ null shells with tied contacts or those that use nodal rigid bodies defined on boxes and finally connect rigidly any nodes that are found within the box.

Intermodular connections in the Modular Environment can be used to generate all possible FE-representations. However, the use of the Smart Assembly methodology, that eliminates the need for loading the individual includes in all cases, reduces the need for employment of sophisticated and complex FE-representations.

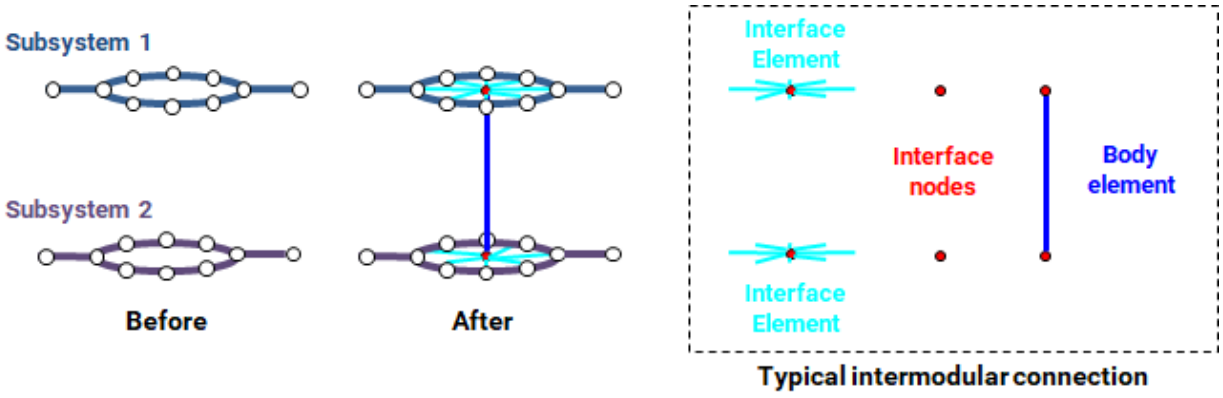
This chapter describes the main assembly-related processes that are managed in the Modular Environment:

- Creation of intermodular connections
- Organization of intermodular connections in the Simulation Run structure
- Realization of intermodular connections and Smart Assembly

6.1. Point Connections

6.1.1. Anatomy of an intermodular point connection

A typical intermodular point connection is shown in the image below:

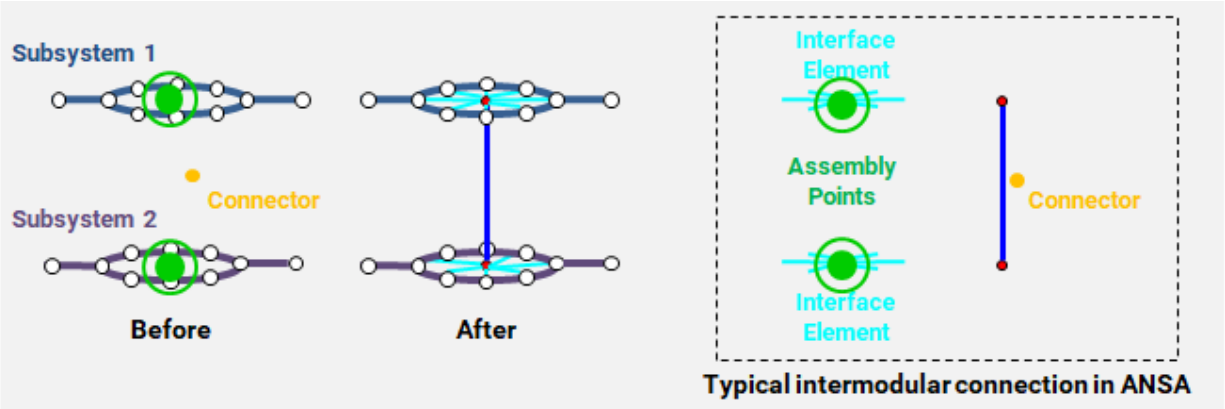


The characteristics of this connection are:

- Two Subsystems (and rarely more than two) are connected with a **body element**, which connects the **interface nodes** that reside in the Subsystems.
- The interface nodes are usually connected to geometric features of the Subsystem through **interface elements**. Quite often, interface nodes mark nodes on the Subsystem geometry, e.g nodes of shell elements or end-points of beams.

In the Modular Run Environment:

- Interface nodes are marked with **Assembly Points**.
- The Assembly Points belong to the modules (i.e. Subsystems or Library Items).
- The body element can be created with a **Connector Entity**.
- The Connector Entity either belongs to a **Connecting Subsystem**, that exists at the same level with the Subsystems it connects, or to the higher level Model Browser Container, that is either the Simulation Model or the Loadcase.
- The interface elements can be created from the Assembly Points directly.
- For the usual case where the interface point of an intermodular connection is located on a screw or a weld-nut, it is possible to use a **bolt connection** in order to generate the interface element together with the **Assembly Point**.
- The creation of interface nodes and interface elements is carried out on Subsystem level while the creation of the body element is carried out on assembly level.



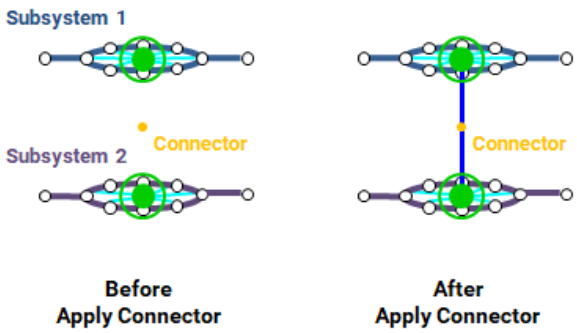
6.1.2. Main types of point connections

The Modular Environment supports the vast majority of point connection FE-representations that are used in actual practice. The most typical examples are:

- RBE2 - CBAR/CBEAM/CBUSH/CELAS - RBE2
- RBE2 - RBE2 - RBE2
- RBODY - SPRING - RBODY
- RIGID PROP - *CONSTRAINED_RIGID_BODY - RIGID PROP, a.k.a. Rigid patches
- TIED CONTACT - NULL SHELLS

These FE-representations can be grouped in two main categories depending on whether the connector element is the “driving” entity that dictates the shape and arrangement of the interface elements or not.

Type A: Connections where the shape of the connector elements stems from the location of the interface points

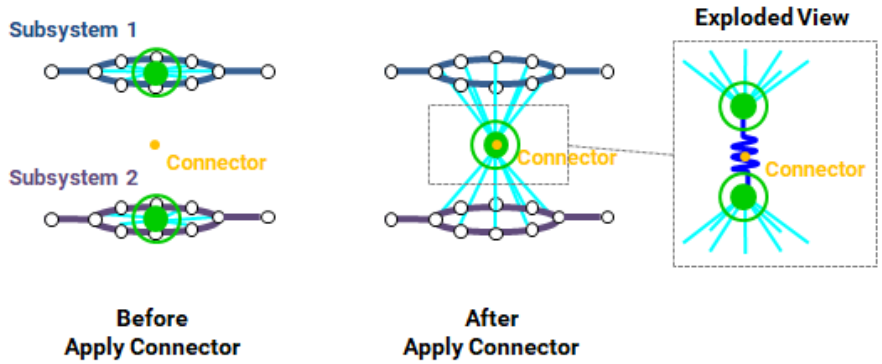


In this case, applying the connector creates a new element that uses the existing nodes that were marked with Assembly Points.

Thus, the length of the connector entity stems from the location of the Assembly Points.

One could say that in this case the Assembly Points are the “drivers” and the connector is a “follower”.

Type B: Connections where the shape of the connector elements dictates the location of the interface points



In this case, the connector has certain specifications for its shape. The connector element is a zero-length element. Therefore, the location of the two interface nodes must be matched. One could say that in this case, the Connector is the “driver” and the Assembly Points are the “followers”.

Almost all types of intermodular point connections can be classified in these two categories. In order to easily understand the type of a connection, the following question can be made:

With conventional includes management, can the includes be completely ready for model assembly independently of one another and not be modified at all during the creation of the connection file?

- If the answer is “yes” it means that the intermodular connection is of **Type A**
- Typical connections of this type are:
  - All the non-zero-length line elements (CBAR, CBEAM, RBE2, SPRING, etc.)
  - The rigid patches used for LS-Dyna
- If the answer is “no”, it means that the intermodular connection is of **Type B**
  Typical connections of this type are:
  - All the zero-length line elements (CBUSH, CONN3D2, *CONSTRAINED_JOINT_ SPHERICAL)
  - LS-Dyna 4-node joints: *CONSTRAINED_JOINT_REVOLUTE, *CONSTRAINED _JOINT_CYLINDRICAL, etc.
  - All the rigid connections realized with nodal rigid bodies in LS-Dyna (*CONSTRAINED_NODAL_RIGID_BODY), Pam-Crash, Radioss (RBODY)
  - All connections realized with NULL shell elements and TIED CONTACTs

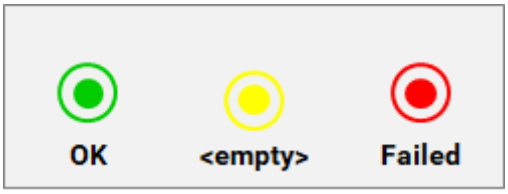
With conventional assembly methodologies, all connections of **Type B** would require the loading in the pre-processor of both mating Subsystems in order to apply the intermodular connection. However, the Modular Run Environment offers a smarter approach that enables the realization of any connection type without the need of loading the mating Subsystems. This whole methodology, known as *Smart Assembly* is described in paragraph 6.1.10. With this methodology, connections of Type B only require a special marking of their Assembly Points during the preparation of the individual connected Subsystems. More information on this marking is given in paragraph 6.1.3.

6.1.3. Marking Interfaces with Assembly Points

The Assembly Points are entities used to mark interface point locations in Subsystems and Library Items. Conventional methodologies mark the interface locations either with specific node ids or specific node names or, in Nastran, with the aid of the field10 node attribute. All three alternatives require a considerable maintenance effort from the model building teams since for each new project, a specification sheet with all interface locations and their proper marking needs to be created and then, for each new version of a module, this specification sheet must be used to mark the interface nodes or verify the marking.

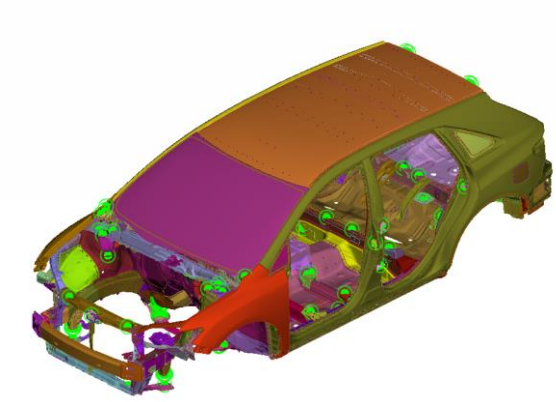
Marking the interfaces with Assembly Points offers the following benefits:

1. There’s a clear visual indication of the interface locations on the model
2. One can see at a glance whether the interface locations are well defined, using the status-based color-coding of Assembly Points

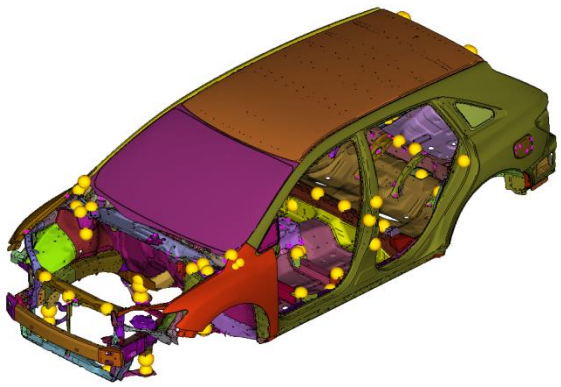


3. The connecting elements are generated by Connector Entities that make reference to A_POINTs. Connectors search for A_POINTs ***by name and proximity***, relieving the user from the tedious task of monitoring the node ids for all interface locations of assembly.
4. The marked locations can be passed to META with the aid of the alc_aux file, which is a text file with ANSA comments that is written by ANSA on Simulation Run output and describes all the interfaces of a

Simulation Run. During the loading of a results file in META, META searches for an alc_aux file with the same name in the same directory and loads it automatically, marking all interface locations.



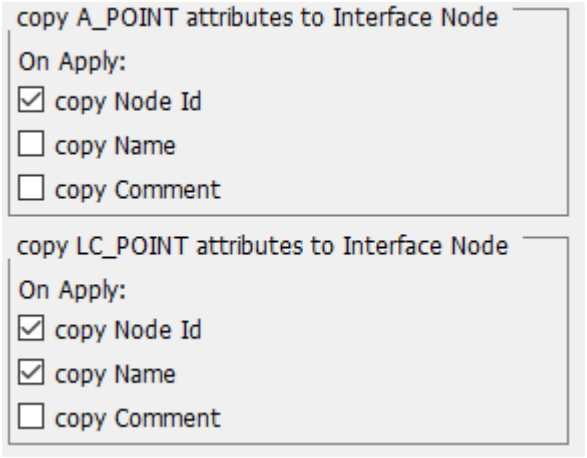
A_POINTs in ANSA



A_POINTs in META

Although node names and ids are not required for assembly, loadcase setup or post-processing in the Modular Environment, it is still possible to control the id and name of the interface node through the Assembly Point, so that models prepared in the Modular Environment are still usable in existing workflows and with 3rd party tools.

This behavior can be activated through Settings>Connections>Connector/Geb/Interface point:



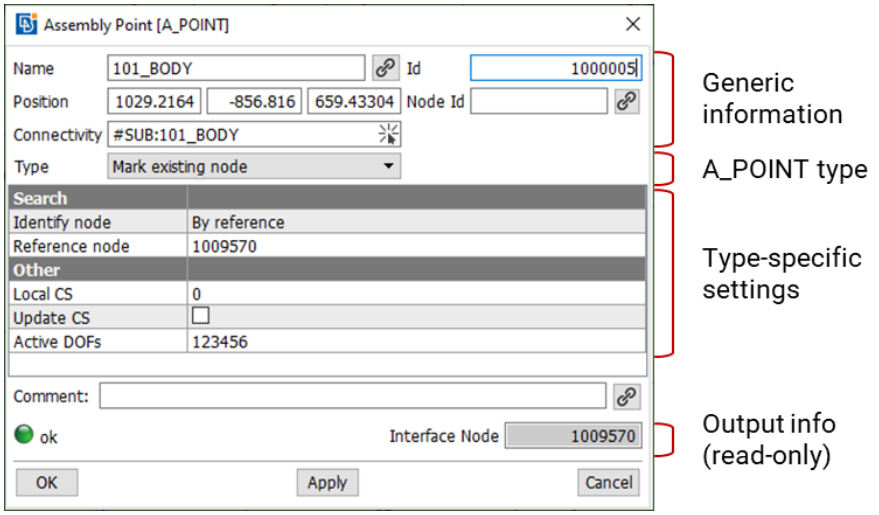
In this form the user can describe the desired behaviour for A_POINTs and LC_POINTs separately. By activating the checkboxes, on A_POINT Apply, any A_POINT attributes among the fields Name, Node Id or Comment will be passed to the underlying interface node.

The corresponding settings in the ANSA.defaults are:

- `copy_A_POINT_id_to_node`
- `copy_A_POINT_name_to_node`
- `copy_A_POINT_comment_to_node`
- `copy_LC_POINT_id_to_node`
- `copy_LC_POINT_name_to_node`
- `copy_LC_POINT_comment_to_node`

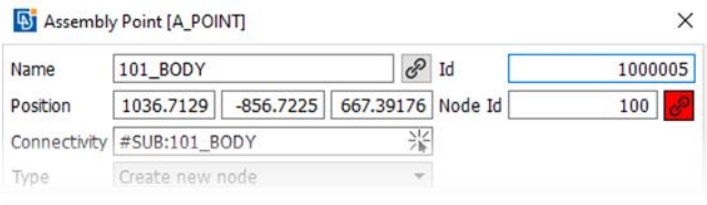
The Assembly Point card has been redesigned in v21. Its main characteristics are described below:

The top area gathers all generic information, like the A_POINT name and position, that are used as the main identifiers of the entity, the connectivity, that holds the module id of the base module the A_POINT belongs to, its Id, that is used as a secondary identifier for compatibility with all other ANSA entities, the Node Id, that can hold the id that should be "pushed" to the identified Interface Node on "Apply".



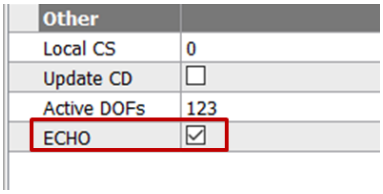
Right below there is the A_POINT type, which determines the settings that will be used for the identification and marking of the interface node. Description of the supported A_POINT types is given in the paragraphs that follow.

Finally, in the bottom area, except for the Comment field, the Status and the Interface Node of the A_POINT are given, both being read-only results of the A_POINT realization.



The toggle buttons with the chain icon next to the Name, Node Id and Comment fields indicate whether the respective A_POINT attribute must be "pushed" to the identified Interface Node on "Apply".

Their default state is controlled from the ANSA.defaults settings described previously and can be toggled individually, in order to change the "copy to node" behavior of particular attributes for a particular A_POINT.

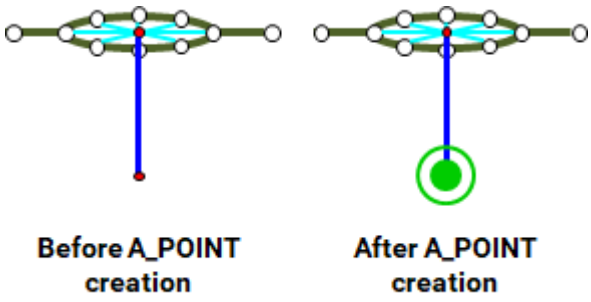


In case of Nastran deck, it is possible to selectively echo the Interface Node of an Assembly Point. Activating the ECHO option in the A_POINT card will add the ECHOON/ECHOOFF keywords before and after the respective GRID in the Nastran keyword file.

Note: The fields as shown in the new card do not necessarily correspond to the names of columns in the Database Browser lists and may not be usable for the "getter" and "setter" script functions. In order to get the Database-Browser-compatible card field names, one can still access the old layout of the A_POINT card by holding down the shift key while editing an A_POINT (i.e. shift + double click in the Database Browser list)

There are four main set-ups for Assembly Points that cover the vast majority of interface entity marking use-cases:

Interface Point Type 1: Mark existing node



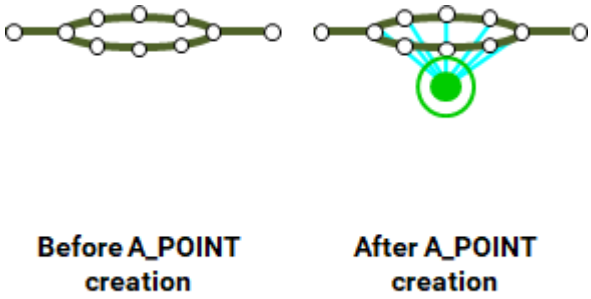
In this case, the user can create an A_POINT that marks an existing node of the model. Such A_POINTs can be easily created through the function A_POINT > New [Node], where the user is prompted to select one or more nodes that will be marked by Assembly Points.

In this case, the A_POINT connectivity is auto-filled with the Module Id of the Subsystem the node belongs to (or the Library Item name)

The key settings of these A_POINTs are:

- **Connectivity**: The Subsystem or Library Item the node belongs to
- **Search**: Identify node by reference, by name or by proximity

Interface Point Type 2: Create new node

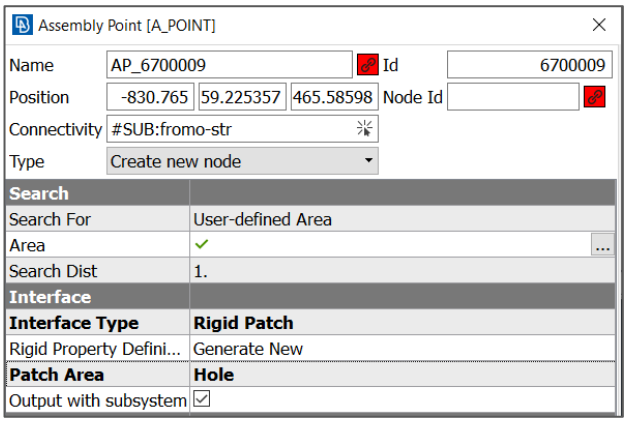


In this case, the user can create an A_POINT at a certain location and search for some geometric features whose nodes will be finally connected with a branch element (RBE2 or RBE3). Finally, the interface node will be a new node that will be created as the central node of the branch entity at the A_POINT location. Such A_POINTs can be easily created through the function A_POINT > New [Location], A_POINT > New [On Holes (Auto)] and A_POINT > New [On COG].

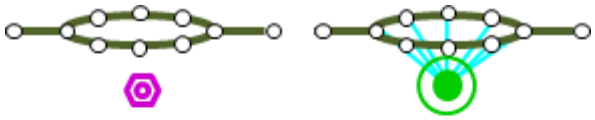
The key settings of these A_POINTs are:

- **Connectivity**: The Subsystem or Library Item where the searched geometric features belong
- **Search**: User-defined area or through Interface Sets
- **Interface**: RBE2, RBE3 or rigid patch

It is possible to generate hole rigid patches as the interface using an A_Point at a certain location, provided that the user has defined a search area or an Interface Set that marks the hole edges or the nodes along the whole circumference. This can be done using the following setup in the A_Point's card:



Interface Point Type 3: From Bolt

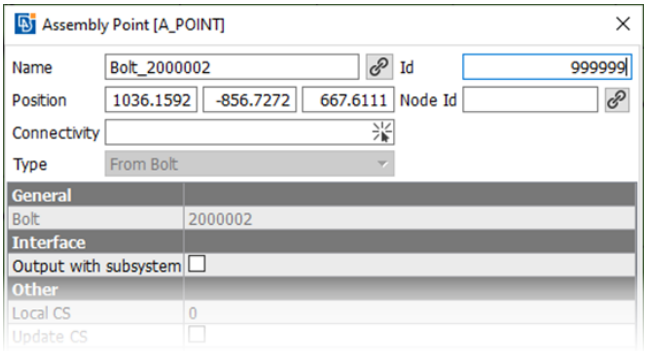


Before Bolt realization

After Bolt realization

Similar A_POINTs can also be created directly by Bolt Connections, as shown below:

In this case, the option **Create Assembly Point** must be activated in the Bolt FE-rep settings.



The auto-created A_POINT has certain settings pre-defined and read-only, as shown on the left.

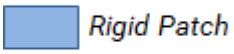
More specifically:

- **Type**: From Bolt (read-only)
- **Bolt**: Id of Bolt Connection (read-only)

Assembly Points of types 2 and 3 (i.e. that create a new node and an interface element) may need to "follow" the specifications of the connector in case of Type B intermodular connections (according to the classification made in paragraph 6.1.2). In such cases, the Assembly Points must be marked as "followers" beforehand, a signal that will be used by the Modular Environment during the realization of intermodular Connectors and also during the output of the keyword files of the connected Subsystems. This marking is achieved through the setting **output with subsystem** in the A_POINT card.

- **output with subsystem**: **Yes** (default value)
  According to the terminology of paragraph 6.1.2, this A_POINT is a "driver" for the intermodular connector generation since its interface node will become one of the endpoints of the intermodular connector.
  It is used for all intermodular connections of **type A**.
- **output with subsystem**: **No**
  According to the terminology of paragraph 6.1.2, this A_POINT is a "follower" for the intermodular connector generation since it follows the specifications dictated by the intermodular connector.
  It is used for all intermodular connections of **type B**.

Interface Point Type 4: Mark interface property

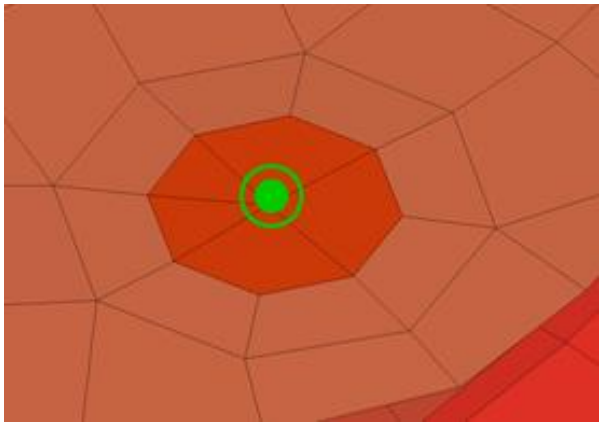


Before A_POINT creation

After A_POINT creation

In this case, the user can create an A_POINT that marks an existing property of the model.

These A_POINTs do not really mark an interface node. They rather mark the position where a patch will be found.



Still, these A_POINTs will report an Interface Node in their card, which will be an arbitrary node of the interface property. However, this node does not play any role for the assembly operations down the road.

The key settings of these A_POINTs are:

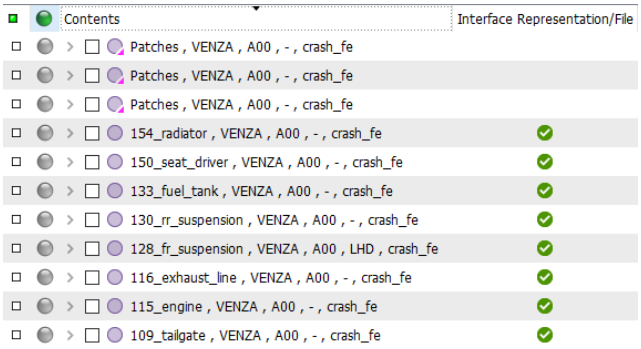
- **Connectivity**: The Subsystem or Library Item the property belongs to
- **Search**: Identify property by reference or by name

6.1.4. Interfaces information when saving modules in DM

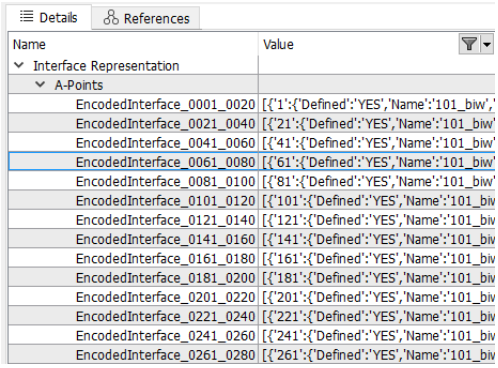
When a Module (Subsystem or Library Item) is saved in DM, the Assembly and Loadcase Points and their FE representations are stored in a separate file named '*interface_representation.ansa*' that is stored as an attached file of the module in DM. The interfaces representation file is always an ANSA file, independently of the File Type of the module.

In case of Assembly Points that are created by bolts, the bolt connection with its FE representation is also included in the interfaces representation.

In case of intermodular Connections of type B (paragraph 6.1.3) the interfaces of the regular Subsystems are stored in DM in their original state, before the realization of the Connector.



In the DM Browser, identifying modules that have an Interface Representation file is easy, with the aid of the *Interface Representation/File* column.



Interface information is also available in the form of attributes on the DM Object. These attributes include all information that could be useful for subsequent assembly operations. More specifically:

- Assembly Point Name and Id
- Assembly Point x, y, z
- Assembly Point *defined* status
- Interface Node id

At present, there's no built-in ANSA process that makes use of the metadata information of interfaces.

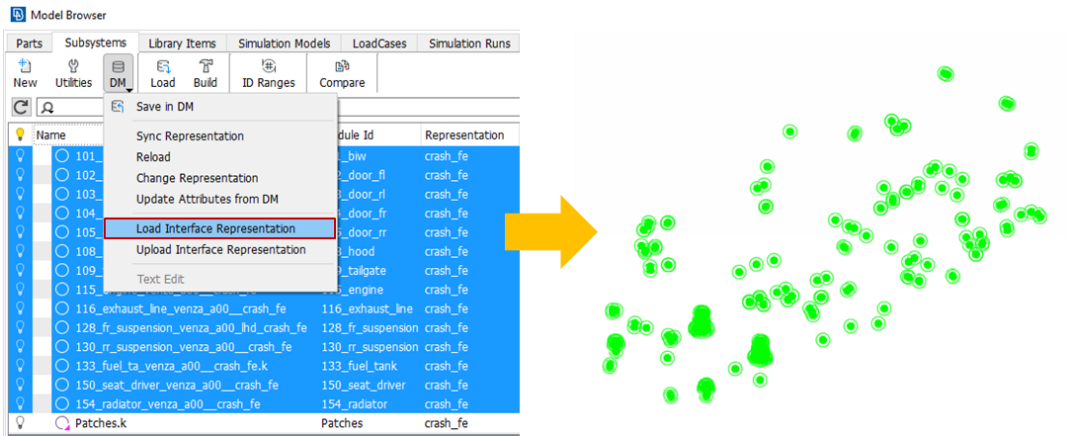
In cases of modules that are copied in DM through the “Add in DM” functionality, and therefore no interface metadata is created for their DM Objects, it is still possible to append interface representation information at a later time through the *DM>Upload Interface Representation* available for Subsystems and Library Items.

6.1.5. Loading interfaces information from DM

Once the interface information has been created in each Subsystem or Library Item and has been saved in DM, it is possible to load in ANSA the interface information alone, in order to prepare the assembly of the compound Model Containers or realize the intermodular connectors.

Low level, this is achieved with the function *DM>Load Interface Representation*

Running this function on a multi-selection of empty modules, will load their interface representations as shown below:



Of course at this point, the Assembly and Loadcase Points, and probably any Bolt Connections that have been loaded, cannot be re-applied, since the actual geometry of the subsystems is not available. However, there’s no need to re-apply these entities for the intermodular assembly, so this is not really a limitation.

Name	Module Id	Representation	Is Interface Representation
101_biw_venza_a00_lhd_crashe_fe.k	101_biw	crash_fe	YES
102_door_fl_venza_a00__crash_fe	102_door_fl	crash_fe	YES
103_door_rl_venza_a00__crash_fe.k	103_door_rl	crash_fe	YES
104_door_fr_venza_a00__crash_fe.k	104_door_fr	crash_fe	YES
105_door_rr_venza_a00__crash_fe.k	105_door_rr	crash_fe	YES
108_hood_venza_a00__crash_fe.k	108_hood	crash_fe	YES
109_tailgate_venza_a00__crash_fe.k	109_tailgate	crash_fe	YES
115_engine_venza_a00__crash_fe.k	115_engine	crash_fe	YES
116_exhaust_line_venza_a00__crash_fe.k	116_exhaust_line	crash_fe	YES
128_fr_suspension_venza_a00_lhd_crash_f...	128_fr_suspension	crash_fe	YES
130_rr_suspension_venza_a00__crash_fe.k	130_rr_suspension	crash_fe	YES
133_fuel_tank_venza_a00__crash_fe.k	133_fuel_tank	crash_fe	YES
150_seat_driver_venza_a00__crash_fe.k	150_seat_driver	crash_fe	YES
154_radiator_venza_a00__crash_fe.k	154_radiator	crash_fe	YES
Patches.k	Patches	crash_fe	NO

Note that ANSA can tell whether a subsystem or Library Item is loaded with its regular or its interface representation and communicates this info through the attribute **Is Interface Representation.**

Loading the regular representation of a module through the Model Browser or when downloading it from the DM Browser:

- If the module is of **File Type ANSA**
  Interface entities come along, since they are saved in the ANSA file of the module
- If the module is of **File Type Solver**
  Interface entities may be contained in the solver keyword file or not, depending on whether the Assembly Point was marked as an intermodular connection “driver” or as a “follower”, according to the terminology of paragraph 6.1.2. The interface entities written in the solver keyword file are:

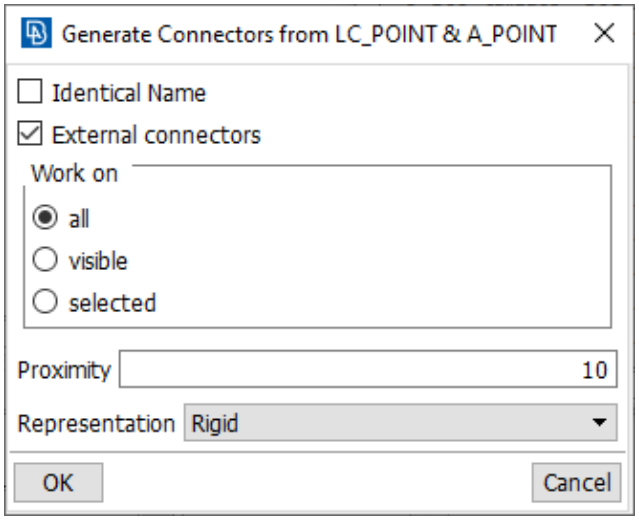
- The solver keywords:
  - Interface node
  - Interface elements, in case of interface points of type 2
- The ANSA comments that describe the Assembly and Loadcase Points

No matter if the interface entities are contained in the solver keyword file, the interface representation ANSA file is automatically merged after the download of the solver keyword file of a module from DM.

This behavior is controlled through the DM setting shown on the left, which is active by default (ANSA.defaults variable: `auto_load_interface_repr_file`)

6.1.6. Creating connecting elements with Connectors

The connecting elements of intermodular connections are typically generated by Connector entities. The Assembly Points of neighboring Subsystems/Library Items can be used to facilitate the creation of Connector Entities through the *New > From A_POINTS (Auto)* function available in the CONNECTOR ENTITY list. This function creates Connectors by matching existing Assembly Points by proximity.



In the *Generate Connectors from LC_POINT & A_POINT* window the user can specify which Assembly Points must be taken into account for the generation of connectors. The user can constrain the connector creation only between A_POINTs that:

- Have identical name
- Belong to different Model Containers
- Are visible or selected

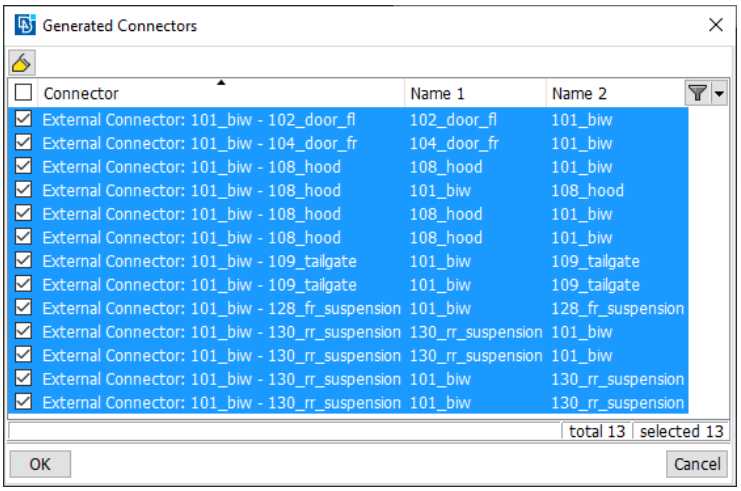
For the identification of mating A_POINTs, the user needs to define a **Proximity distance**. In the end, a **Representation** must also be defined.

The list of supported representations is different for each deck. The options used more often are:

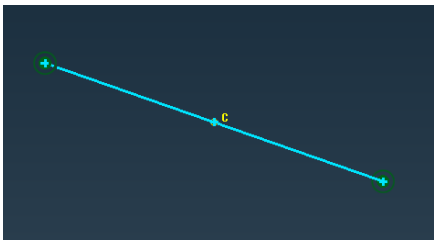
- Nastran: Rigid, CBUSH
- Ls-Dyna: Rigid, Constrained Rigid Body
- Pam-Crash: Rigid, PamCrash SPRGBM
- Radioss: Rigid, Radioss Spring
- Abaqus: Rigid

However, if the required connecting element type is not among the ones supported out-of-the-box, the user can select the option **None** and then, after the creation of the Connectors, set-up a custom body representation with the option **From File**.

Pressing **OK**, the function shows a preview of the connectors to be generated.



Isolating a connector, the user can see the A_POINTs that were paired and the preview of the connecting element.



Note that this function can create a connector that connects more than 2 A_POINTs.
In the end, the function reports the remaining "orphan" A_POINTs in the *ANSA Info* window.

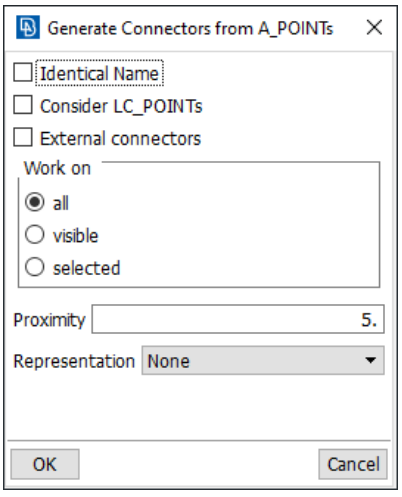
```
CONNECTOR New>From A_POINTs (auto): Unused A_POINTs are:
11032627,11032629,11032633-11032637,11032640-11032655
```

The string of ids is formatted in a way that can be directly pasted in the filter field of the A_POINTs list in order to isolate the orphan A_POINTs in the list and graphically.

Intermodular Connector Entities can also be created directly on a Simulation Model, through the option Actions>Create Connectors.

In this case, all Assembly Points contained in the modules of the Simulation Model are considered as candidates for matching.

The Connector Entities generated through this process are properly configured:



- **Location**: Their (x, y, z) is automatically set to the geometrical center of the A_POINTs being connected
- **Orientation**: Their (dX_x, dX_y, dX_z) vector is automatically defined along the vector between the two A_POINTs
- **Connectivity**: Their connectivity fields are automatically filled with the Module Id or Name of the connected Subsystems or Library Items respectively
- **Search**: Their search is switched automatically to **Assembly Point** and they are configured to identify the A_POINT by name and by reference
- **Interface**: Their interface is set to **From A_POINT,** indicating that any interface element involved in the intermodular connection will be contributed by the Assembly Point
- **Representation**: Their body element is set according to the Representation option selected in the *Generate Connectors from LC_POINT & A_POINT* window

Remarks

It is advisable to use this function starting with a small proximity distance, in order to first identify matches with high certainty and start increasing the proximity gradually. A_POINTs that are already used by a Connector are automatically excluded from the search.

6.1.7. Creating connecting elements manually

Although the recommended approach for the creation of point intermodular connections is the use of Connector Entities, there are cases where the use of Connectors is not feasible and the manual creation of connecting elements cannot be avoided. The Modular Environment considers as *connecting element* any element that does not belong to the same module with its nodes.

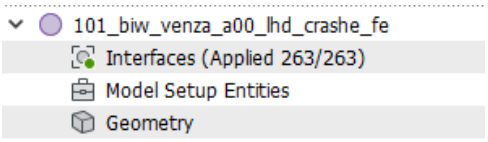
One can come across this case during the migration of existing models that reference connecting includes (i.e. includes that contain connecting elements). In these cases, the plugin “From Includes to Model Browser” offers the option to mark these external references with Assembly Points automatically.

Connecting elements between Assembly Points trigger the application of a particular treatment during model decomposition and recomposition. More information is provided in paragraph 6.1.8.

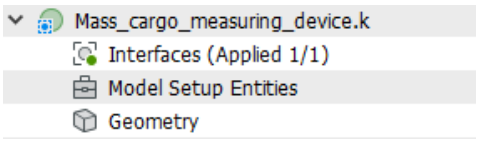
6.1.8. Association of Assembly Points with Model Containers

The parent module of Assembly and Loadcase Points is assessed automatically based on their connectivity. This means that the user should not attempt to assign the Assembly and Loadcase Points to particular Subsystems or Library Items with drag and drop.

The interfaces assigned automatically to a module are displayed under the ***Interfaces*** container.



Subsystem Interfaces



Library Item Interfaces

The automatic assessment of the parent module is based either on the Assembly Point connectivity field, or, in case of Node Reference search pattern, based on the ownership of the referenced node.

6.1.9. Assignment of Connecting Elements to Model Containers

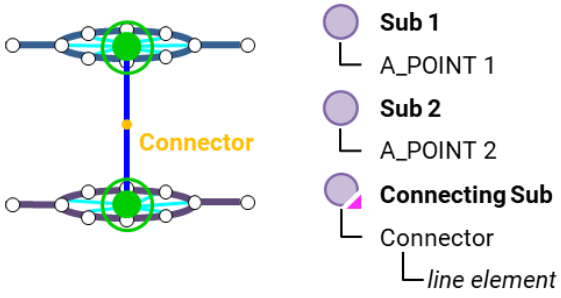
Most of the time, Connecting Elements are assigned to Connecting Subsystems. These Connecting Subsystems are assigned to the same Simulation Model with the Subsystems they connect. However, there are also some other, rare cases, where the Connecting Elements need to be assigned to the higher level compound container (e.g. to the Simulation Model, or the Loadcase or even the Simulation Run).

All alternative options for the organization of intermodular connection elements are described in paragraph 6.4.

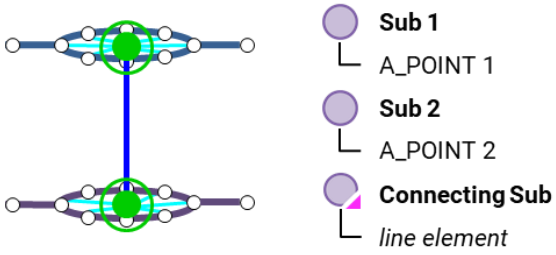
6.1.10. Decomposition and composition of a point intermodular connection

Once the interface locations of a point intermodular connection have been marked with Assembly Points, it is possible for ANSA to break down the connection during modular Save or Unload (decomposition) and reassemble it automatically on Load (composition).

In the sketches below we see two typical examples of inter-modular connections between two regular Subsystems. In both cases the connecting elements are stored in a Connecting Subsystem.



Case 1:
Connecting element is generated by a Connector Entity



Case 2:
The connecting element is created manually

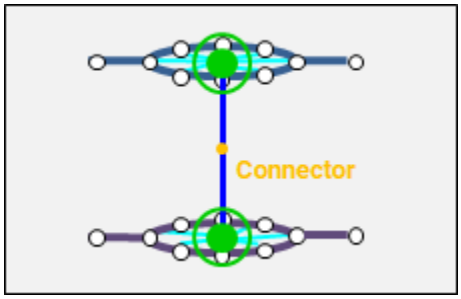
The procedures described below apply to both cases.

Decomposition during modular Save

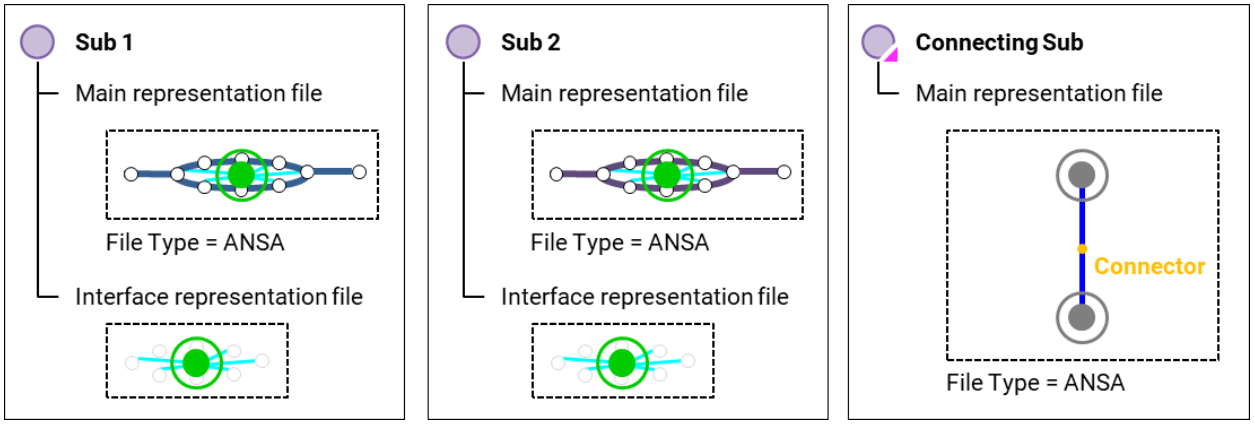
Given that the interface points have been marked with Assembly Points, during Save of the Connecting Subsystem as an ANSA file, ANSA detects that the end-points of the connecting elements are marked with Assembly Points that do not belong to the connecting subsystem and generates automatically *undefined* Assembly Points in the saved file. These undefined Assembly Points mark the nodes at the end-points of the connecting elements, indicating that these nodes *should not really exist in the connecting file*. At the same time, the nodes are marked as auxiliary (AUXILIARY=YES) so that they wouldn't be written out in a solver keyword file in case the Connecting Subsystem is output and, at the same time, indicating that during id conflicts with other nodes, they should be offset in all cases.

During decomposition, the Connector entity maintains its Status in the Connecting Subsystem. However, since the Connector still references the regular Subsystems in its connectivity fields, it is not possible to be successfully reapplied without loading the actual interface points of the regular Subsystems.

Before modular "Save"

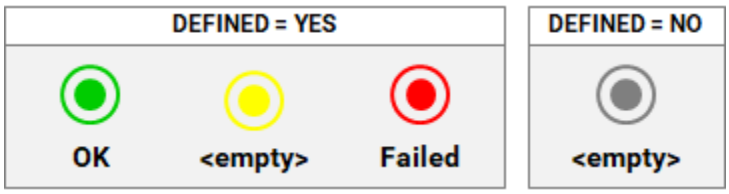


After decomposition



The existence of undefined Assembly Points allows the user to open the Connecting Subsystems and make manual modifications in the connecting elements without having to care about the node ids at their end-points. As long as the end-points are marked with undefined Assembly Points, the intermodular assembly can be rebuilt at any time.

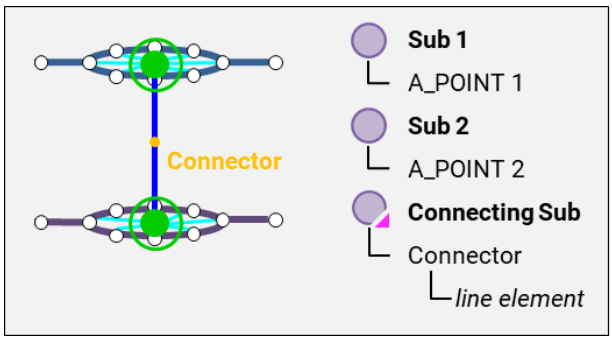
Note: Undefined Assembly Points have grey color and empty status at all times.



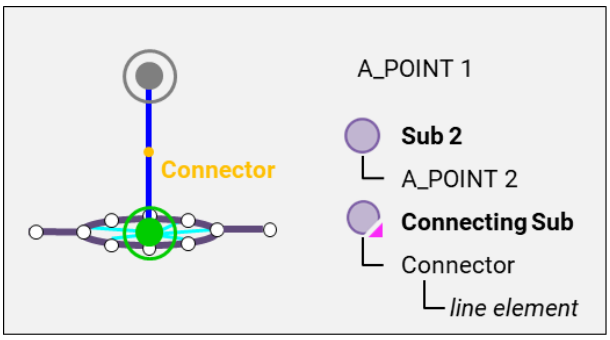
Decomposition during Unload

Upon unloading a module (i.e. Subsystem or Library Item) that contains defined interface points used by entities that belong to a different module, ANSA automatically creates undefined interface points. These undefined interface points enable the automatic re-assembly of the affected entities when the module is loaded back.

Before "Unload"



After "Unload"



Composition during Load - Smart Assembly

The reassembly that takes place during "Load" is based on the inherent capability of Assembly Points to get merged under certain circumstances. Two Assembly Points get merged when:

- The one Assembly Point is defined and the other is undefined
- The two Assembly Points have the same name or have the same name and position, based on the value of the ANSA.defaults variable ***merge_A_points*** which can get the values ***equal name*** and ***equal name and position.***

During the merging of an undefined with a defined Assembly Point:

- The Node Id / Property Id marked by the undefined Assembly Point is replaced by the Node Id / Property Id marked by the defined Assembly Point
- If during this process, indeed, the Node Id / Property Id marked by the undefined Assembly Point gets updated, the Connecting Subsystem is marked as *modified* (magenta), indicating that a new version of the connection file must be created
- Otherwise, if the Node Id / Property Id marked by the undefined Assembly Point remains the same, the Connecting Subsystem is not marked as *modified* (magenta), indicating that the existing version of the connection file can still be used.

The above functionality applies also in case that there are more than one undefined Assembly points that are matched with a defined Assembly Point. In such case the Node Ids /Property Ids of all the undefined Assembly Points are replaced with the Node Id / Property Id marked by the defined Assembly Point.

With this functionality, the only data required in order to update the connection file is the collection of *defined* Assembly Points that reside in the Interface Representation files of Subsystems and Library Items, and the *undefined* Assembly Points, that reside in the connecting subsystems. Thus, we could say that the *Smart Assembly* process in ANSA is a two-step process where:

1. The Interface Representation files of all regular Subsystems are loaded, through *DM>Load Interface Representation*
2. The standard ANSA files of all connecting Subsystems are loaded, through *Load*

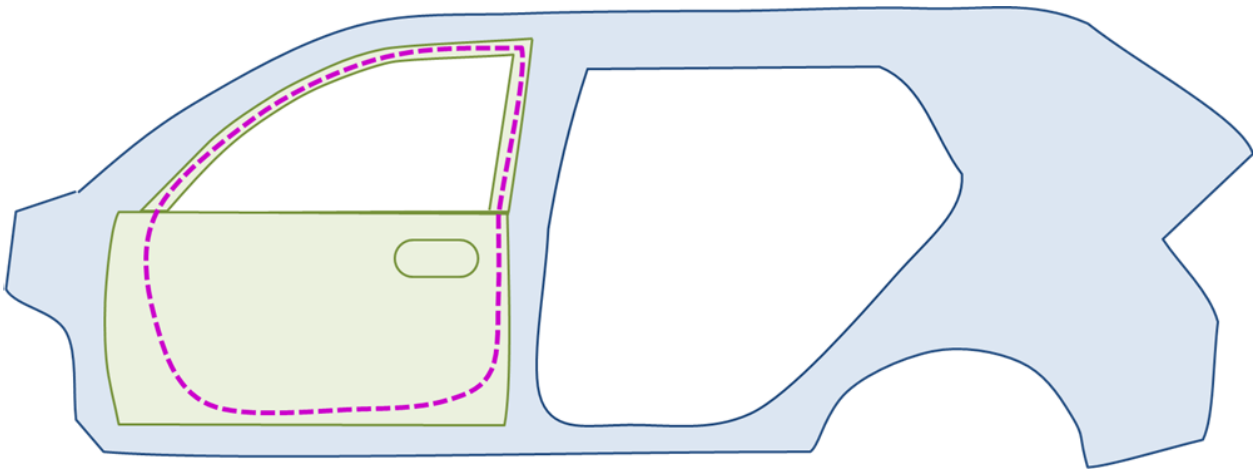
Exactly these steps, with this order, are implemented by the **Load Base Modules** Build Action available on Simulation Models and Loadcases.

After composition, and in order to verify the assembly, the user should check that there are no remaining Assembly Points with “undefined” status. This is definitely an indication of error. Additionally, the user could check that there are no “orphan” defined Assembly Points. An “orphan” Assembly Point could indicate a potential problem, but, in most of the cases it’s a normal situation. The check for remaining undefined Assembly Points is included in the **Check Assembly Status** Build Action available on Simulation Models and Loadcases.

6.2. Continuous Connections

6.2.1. Anatomy of an intermodular continuous connection

A typical continuous intermodular connection is shown below:



The characteristics of this connection are:

1. Two Subsystems are connected with a series of **interface-body-interface** element combinations (e.g. RBE3-HEXA-RBE3, RBE3-CBUSH-RBE3, etc.) which connect the **surface elements** (shells or solid facets) that reside in the Subsystems.
2. These interface-body-interface elements are generated by a **connection curve** like Adhesive Line or Seam Line or by a **connection face** like an Adhesive Face

In the Modular Run Environment:

- There's nothing equivalent to the Assembly Point that could be used for the realization of a continuous connection
- The interface-body-interface elements can be created with a **Connection Curve** (e.g. Adhesive Line, Seam Line, etc.) or a **Connection Face** (i.e. Adhesive Face)
- The Connection Curve or Face belongs to a **Connecting Subsystem** that exists at the same level with the Subsystems it connects.
- During Smart Assembly, the Build Process decides automatically, based on compatibility check between Connecting and Regular Subsystems, whether the full or the Interface representation of the mating modules needs to be loaded.

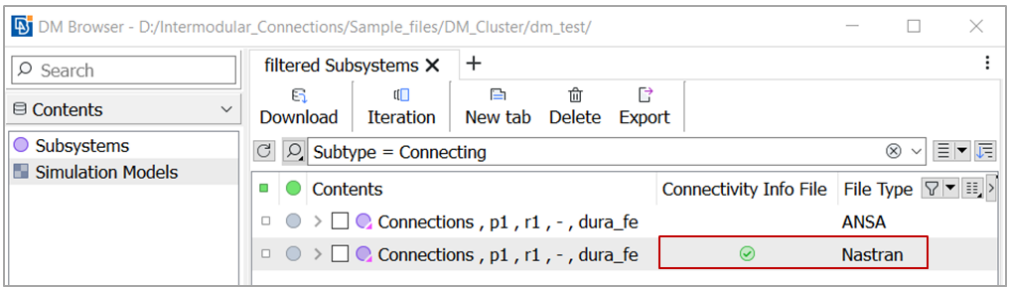
6.2.2. Connectivity information in DM

The Intermodular Connections functionality is by default inactive. In order to enable it, modify the DM setting "intermodular connectivity links" in the dm_structure.xml as shown below:

```
<DM Setting name = "intermodular connectivity links" value "YES"/>
```

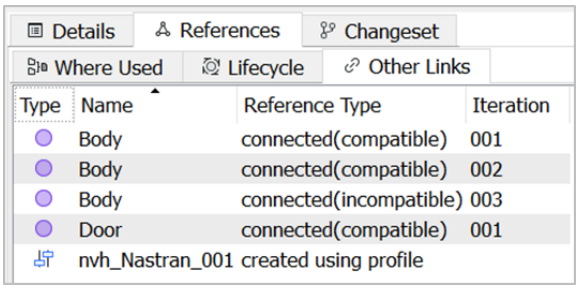
Note: With SPDRM back-end, this setting is defined in the server configuration file *taxis.conf*.

When a Connecting Subsystem that contains Connection Entities is saved in DM as a solver file, the information of the local geometry of the Regular Subsystems used by the FE representations (e.g. nodes, shells position), as well as the information related to external entities of the FE representation (e.g. Node Ids, PIDs) is stored as metadata in an attached file in DM. Additionally, Intermodular Connectivity links are created between the Connecting Subsystem and the Regular Subsystems it connects with Connection Entities.

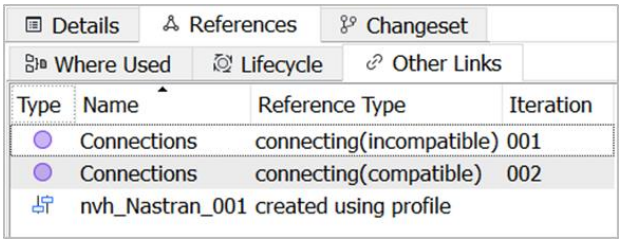


In the DM Browser, Intermodular Connectivity links can be easily identified when selecting either a Connecting or a Regular Subsystem from the **References > Other Links tab**. These links reveal which modules are connected and come in two sub-types:

- A *compatible* Connectivity Link is created between a Connecting Subsystem and Regular Subsystems when the solver file of the Connecting Subsystem can be used "as is" in an assembly process where all these Subsystems participate
- An *incompatible* Connectivity Link is created between a Connecting Subsystem and Regular Subsystems when the solver file of the Connecting Subsystem cannot be used "as is" in an assembly where all these Subsystems participate. An incompatible link is an indication that the Connections within the Connecting Subsystem must be re-applied in order to produce a new, compatible, Connecting Subsystem.



Having selected a Connecting Subsystem in the top list



Having selected a Regular Subsystem in the top list

The Intermodular Connectivity Links are always created between the solver file of the Connecting Subsystem and the Regular Subsystems. If an ANSA representation file is saved in DM for the Regular Subsystems, it is always the one used in the link. Otherwise, the link is created with the solver representation of the Regular Subsystems.

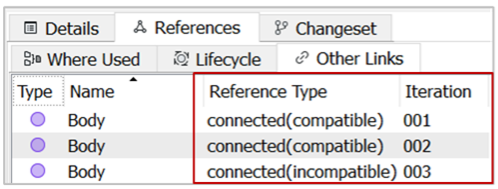
Linked DM objects when the primary file type for Regular Subsystems is ANSA

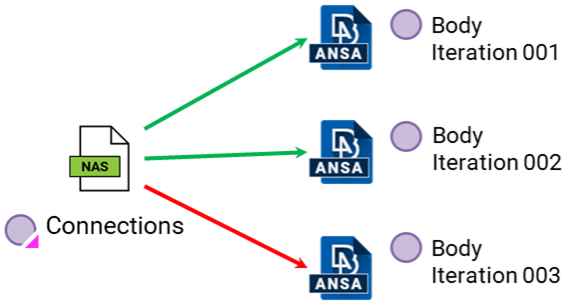
Linked DM objects when the primary file type for Regular Subsystems is solver file

6.2.3. Inheriting Connectivity Links

The Intermodular Connectivity Links are first created the moment a Connecting Subsystem is saved as a solver file in DM. From then on, every time the user saves in DM a new version of a Regular Subsystem whose parent has Intermodular Connectivity Links in DM, existing links of the parent are inherited and are assessed as far as their compatibility is concerned:

- If the new version of the Regular Subsystem has modifications that do not affect the Intermodular Connections, the link is inherited as *compatible*.
- If the new version of the Regular Subsystem has modifications that affect the Intermodular Connections, the links is inherited as *incompatible*.





6.2.4. Composition of a continuous intermodular connection

Without Intermodular Connections, the assembly that takes place during “Load” is based on the loading of the Interface Representations of regular Subsystems and the standard ANSA files of all connecting Subsystems. A connection file (i.e. the solver representation of a Connecting Subsystem) will only have to be re-generated in case the connecting Subsystem after “load” is marked as *modified* (magenta). This marking indicates that since the connection file was created, there were changes on the regular Subsystems that affected the interface locations in a way that affects the connection file. This whole process is implemented in the **Load Base Modules** Build Action of Simulation Models and Loadcases.

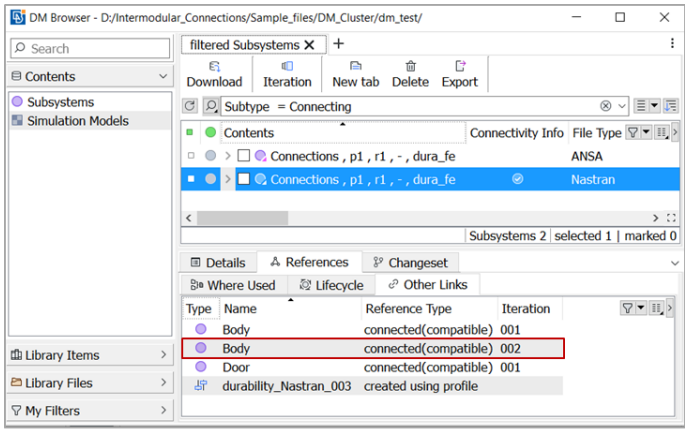
With Intermodular Connections present, if a connection file needs to be re-generated because of changes on the regular Subsystems that affected the interface locations, loading the Interface Representation of regular Subsystems won’t be enough, as the Intermodular Connections require the full representation of the regular Subsystems to be re-applied. Thus, in the presence of Intermodular Connectivity Links on the connecting Subsystems, the **Load Base Modules** Build Action will decide whether the Interface or the Full Representation of a Subsystem needs to be loaded, after assessing the compatibility of the regular with the connecting Subsystems through the link type: Compatible or Incompatible.

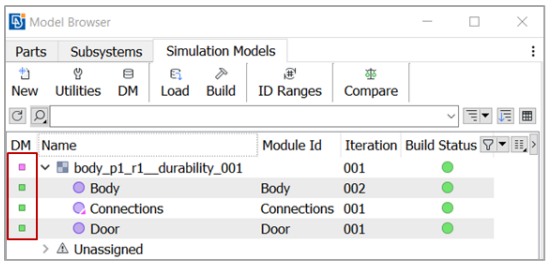
Once an Incompatible link is detected, the Build Action will load the respective regular Subsystems as well as their mating Subsystems with their full FE-representation and make sure to re-apply the related Intermodular Connections.

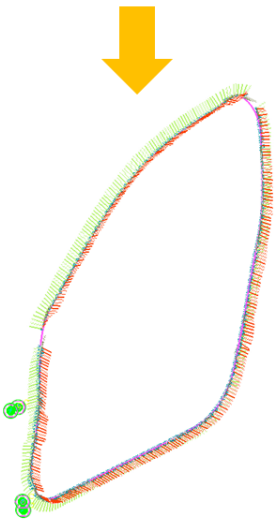
In the example below, the assembly of Door with the Body Subsystem is explored. In this case, the Door is connected to the Body with a combination of point connections (Connector Entities) and curve connections (Adhesive Line). The first version of the connection file (Iteration 001) holds compatible connectivity links to the regular Subsystems of the Body and Door (Iterations 001). Two new versions of the Body are created: Iteration 002, with no modification in the connection area and Iteration 003 with some changes in the connection area.

- In case the solver file of the Connecting Subsystem is still compatible with the current state of the Subsystems being connected, only the Interface representation of the Subsystems will be loaded.
- In case the newer version of the regular Subsystems requires the re-realization of the FE representation of the continuous connections, then the full representation of all Subsystems will be loaded in the model.

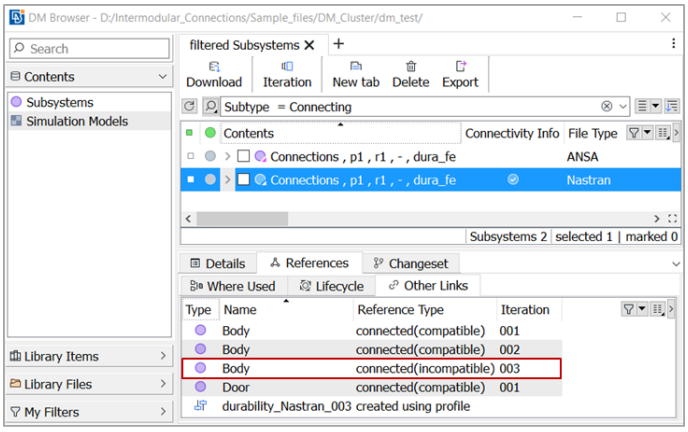
Case: New Subsystem Iteration is compatible with the existing connection file. Connection file reuse.

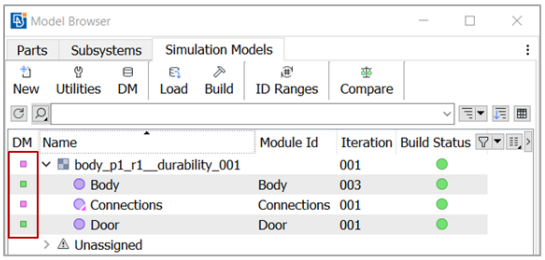


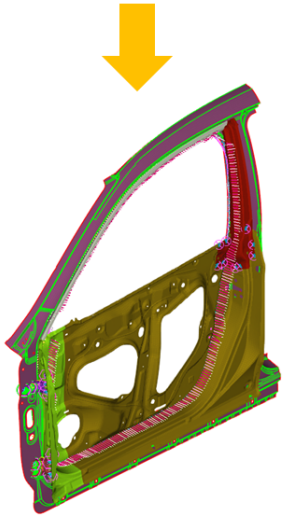




Case: New Subsystem Iteration is incompatible with the existing connection file. Connection file regeneration.







On the left, the Connecting Subsystem will not lose its “Up to date” DM Update Status and the compound Model Container is ready to be saved in DM. On the right, since it is necessary to reapply the FE representations of the connections, the DM Update status of the Connecting Subsystem will turn to “Modified”.

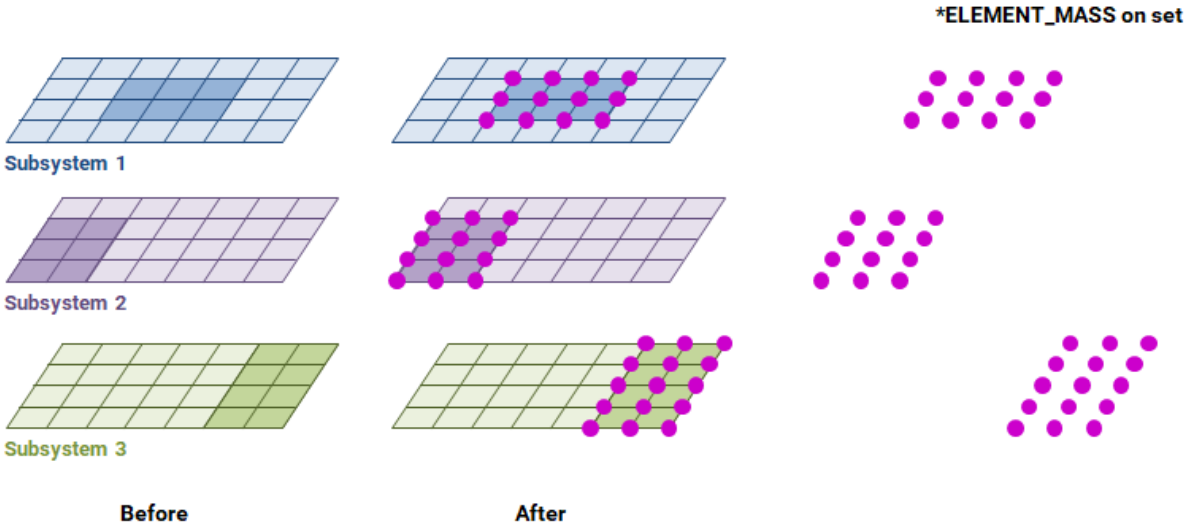
This whole process is managed by the **Load Base Modules** Build Action available on Simulation Models and Loadcases.

6.3. Creation of set-based Connections

6.3.1. Anatomy of an intermodular set-based connection

This category of intermodular connections is not strictly used for connection purposes. It's rather used in order to create a "union" of sets that come from the different modules.

A typical assembly of sets is shown in the image below:

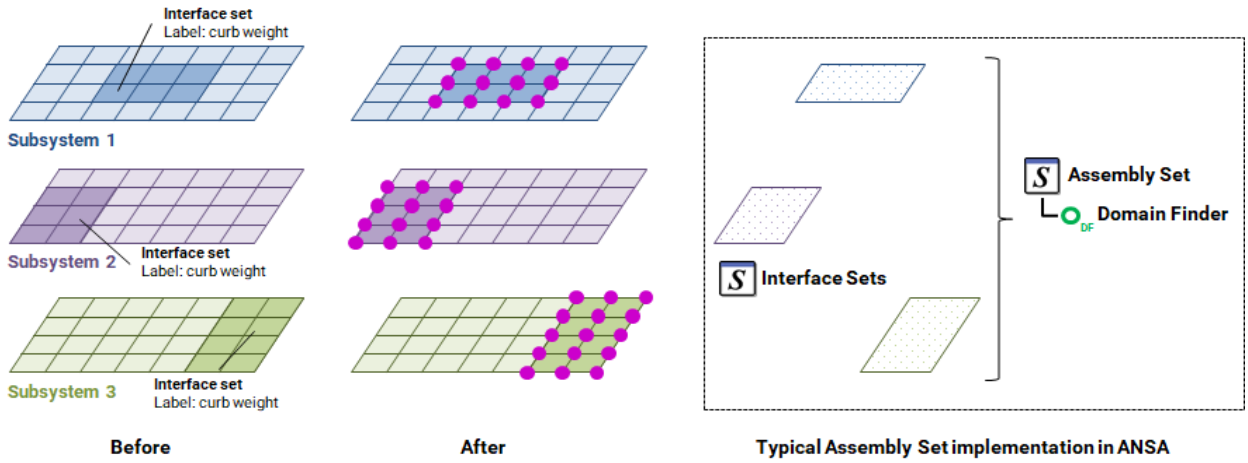


The characteristics of this connection are:

1. Each module contributes nodes, elements or properties by adding them to a **labeled interface set** that resides in the module
2. In order for an entity (e.g. mass, contact, time history markers, boundary condition, etc.) to make reference to a single set that contains all the entities contributed by each module, an **assembly set** is created, that collects all the subsets and is finally used by the entity

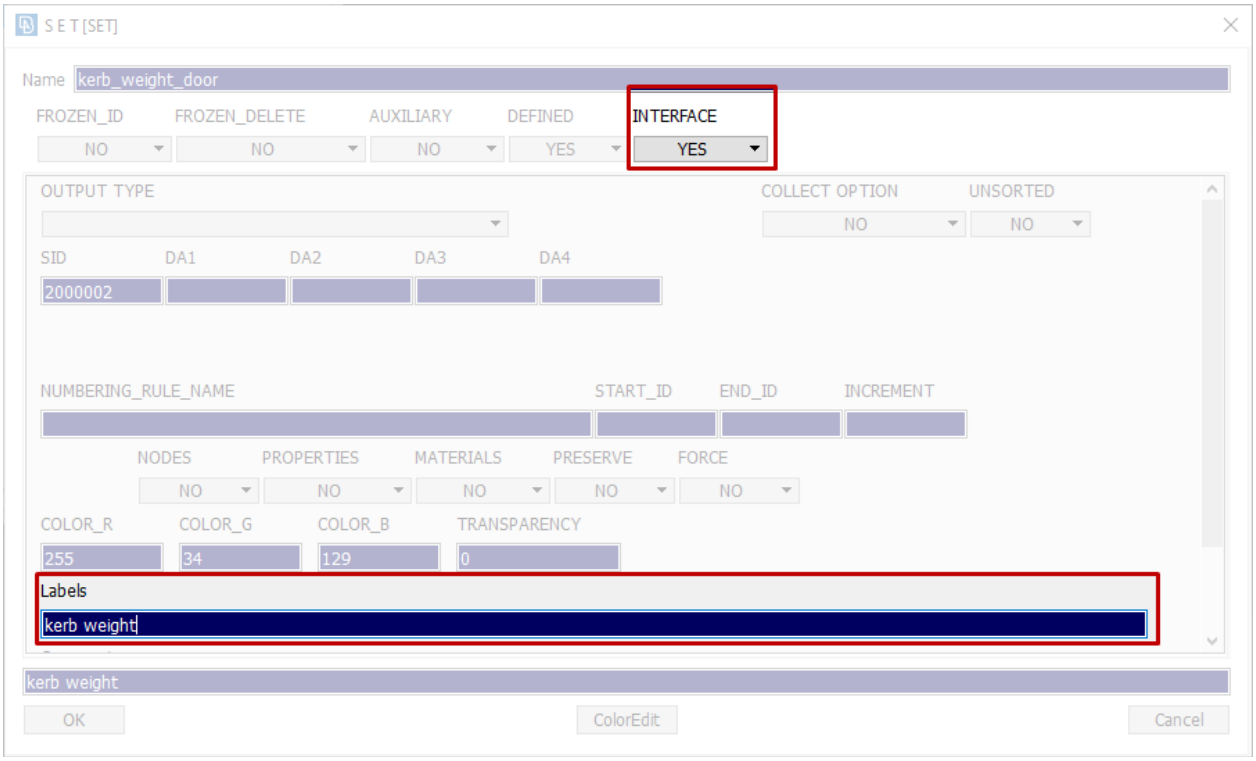
In the Modular Run Environment:

- Interface sets are standard sets marked with the flag **INTERFACE=YES** and labeled with one or more labels that are defined comma-separated in the field **labels**
- The interface sets belong to the modules (i.e. Subsystems or Library Items)
- Assembly sets are standard sets that contain a **Domain Finder**, which is an ANSA entity used to search for other entities, in this case interface sets. Whatever entities are found from the Domain Finder are automatically considered contents of the Assembly Set.
- The entity that will "consume" the Assembly Set (e.g. the mass element in the example above) is most of the times created manually, with no need for an ANSA generator
- This entity, together with the Assembly Set either belong to a **Connecting Subsystem**, that exists at the same level with the Subsystems it connects or to the higher level Model Browser Container, that is either the Simulation Model or the Loadcase
- The creation of interface sets is carried out on module level while the creation of the Assembly Set is carried out on assembly level.

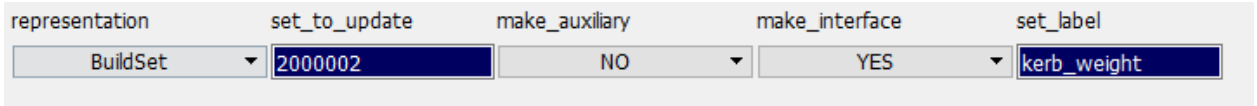


6.3.2. Creating Interface Sets

Any set can be marked as an *interface* set by activating the INTERFACE flag in the set card.

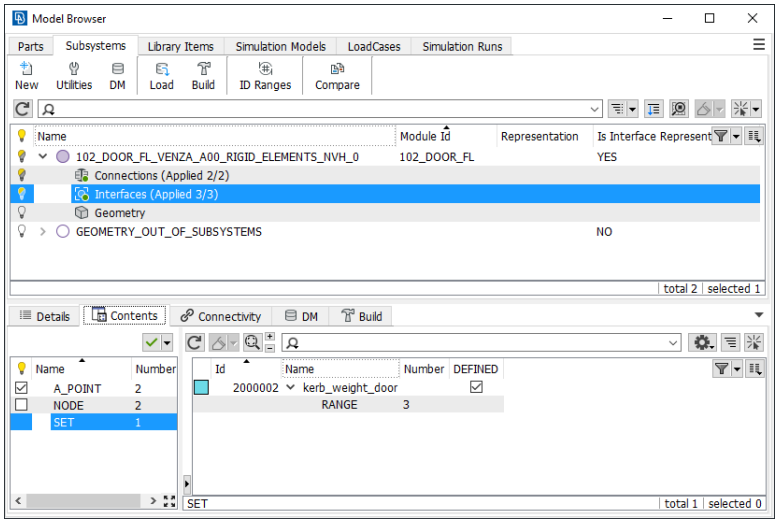


Alternatively, interface sets can be created with a Generic Entity Builder (GEB_SB) by activating the **make_interface** flag, and, optionally, setting a label in the **representation** section of the GEB_SB card.

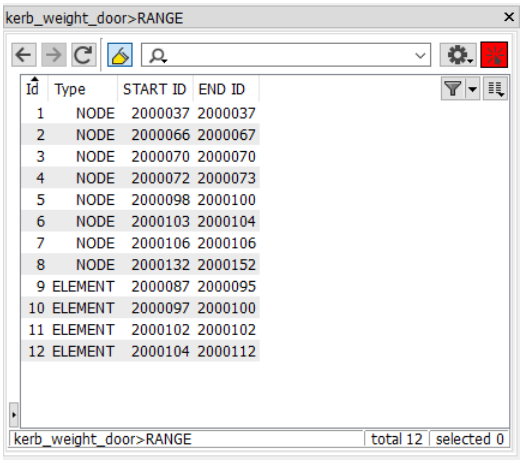


6.3.3. Interface sets information when saving modules in DM

When a Module (Subsystem or Library Item) is saved in DM, the Interface Sets are stored in a separate file named '*interface_representation.ansa*' that is stored as an attached file of the module in DM. The interfaces representation file is always an ANSA file, independently of the File Type of the module.



The sets contained in the interface representation file do not contain their actual contents since this would considerably increase the content of the interface representation file, to the point of making it comparable with the size of the main representation file of the module. However, since, depending on solver, access to the set contents may be required in order for an entity to make reference to the set, these sets are not saved empty.



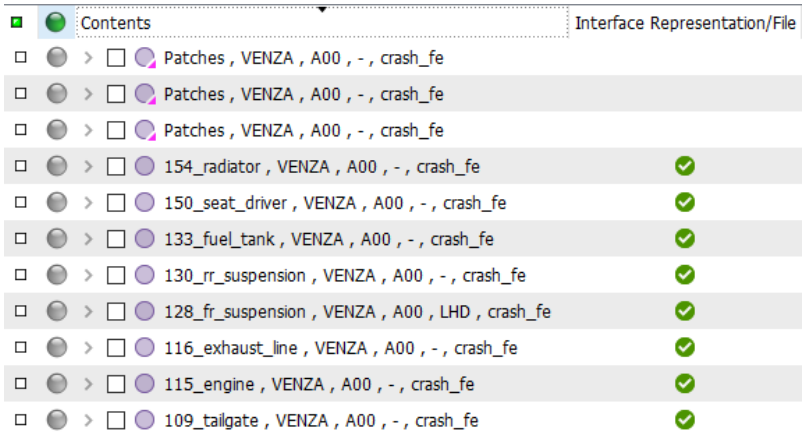
During the creation of this file, depending on the original entity types of the set contents, ANSA writes the set contents in the interface representation without adding an overhead to the size of the representation file.

This is achieved by converting the source contents of the sets into RANGE entities. RANGE entities hold information on entity type and entity ids. ANSA can create ranges of nodes or elements.

Since at the moment of Save the final "consumer" of the set is not yet known, RANGE entities of multiple types are generated, starting from the higher level entity type, dictated by the entities originally added to the set, and going down to the lowest level entity type, the node.

Entities types contained in the interface set defined in the module	**Entities types contained in the interface set defined in the interface representation file**
Properties	RANGE of Type = Property RANGE of Type = Element RANGE of Type = Node
Elements	RANGE of Type = Element RANGE of Type = Node
Nodes	RANGE of Type = Node

This way, the information available in the interface representation file will be usable by any kind of “consumer”. More information on the use of sets is given in paragraph 6.3.6.



In the DM Browser, identifying modules that have an Interface Representation file is easy, with the aid of the *Interface Representation/File* column.

Interface information of sets is also available in the form of attributes on the DM Object. These attributes include all information that could be useful for subsequent assembly operations. More specifically:

- Set Name and Id
- Set Labels

At present, there’s no built-in ANSA process that makes use of the metadata information of interfaces.

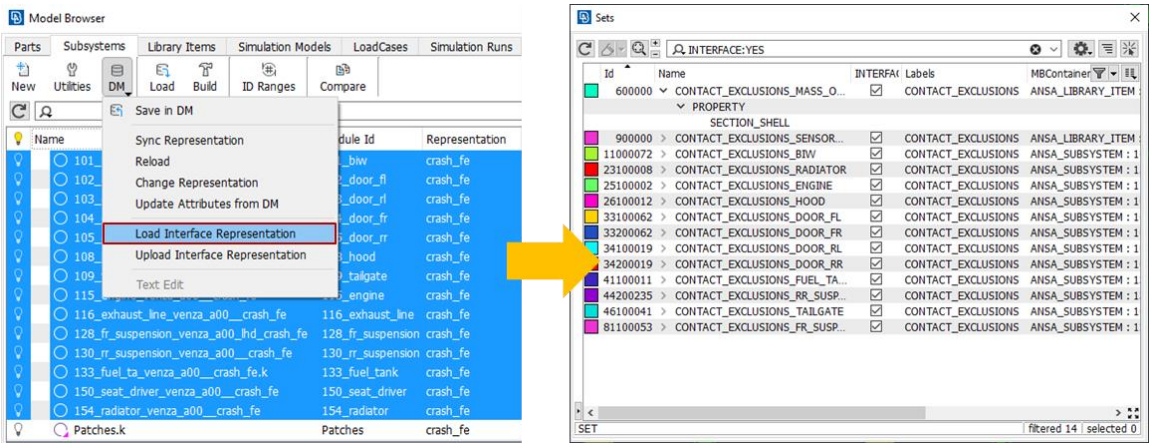
In cases of modules that are copied in DM through the “Add in DM” functionality, and therefore no interface metadata is created for their DM Objects, it is still possible to append interface representation information at a later time through the *DM>Upload Interface Representation* available for Subsystems and Library Items.

6.3.3.1. Loading interfaces information from DM

Once the interface information has been created in each Subsystem or Library Item and has been saved in DM, it is possible to load in ANSA the interface information alone, in order to create assembly sets or use the interface sets directly during Loadcase setup.

Low level, this is achieved with the function *DM>Load Interface Representation*

Running this function on a multi-selection of empty modules, will load their interface representations as shown below:



Name	Module Id	Representation	Is Interface Representation
101_biw_venza_a00_lhd_crashe_fe.k	101_biw	crash_fe	YES
102_door_fl_venza_a00__crash_fe	102_door_fl	crash_fe	YES
103_door_rl_venza_a00__crash_fe.k	103_door_rl	crash_fe	YES
104_door_fr_venza_a00__crash_fe.k	104_door_fr	crash_fe	YES
105_door_rr_venza_a00__crash_fe.k	105_door_rr	crash_fe	YES
108_hood_venza_a00__crash_fe.k	108_hood	crash_fe	YES
109_tailgate_venza_a00__crash_fe.k	109_tailgate	crash_fe	YES
115_engine_venza_a00__crash_fe.k	115_engine	crash_fe	YES
116_exhaust_line_venza_a00__crash_fe.k	116_exhaust_line	crash_fe	YES
128_fr_suspension_venza_a00_lhd_crash_f...	128_fr_suspension	crash_fe	YES
130_rr_suspension_venza_a00__crash_fe.k	130_rr_suspension	crash_fe	YES
133_fuel_tank_venza_a00__crash_fe.k	133_fuel_tank	crash_fe	YES
150_seat_driver_venza_a00__crash_fe.k	150_seat_driver	crash_fe	YES
154_radiator_venza_a00__crash_fe.k	154_radiator	crash_fe	YES
Patches.k	Patches	crash_fe	NO

Note that ANSA can tell whether a Subsystem or Library Item is loaded with its regular or its interface representation and communicates this info through the attribute **Is Interface Representation.**

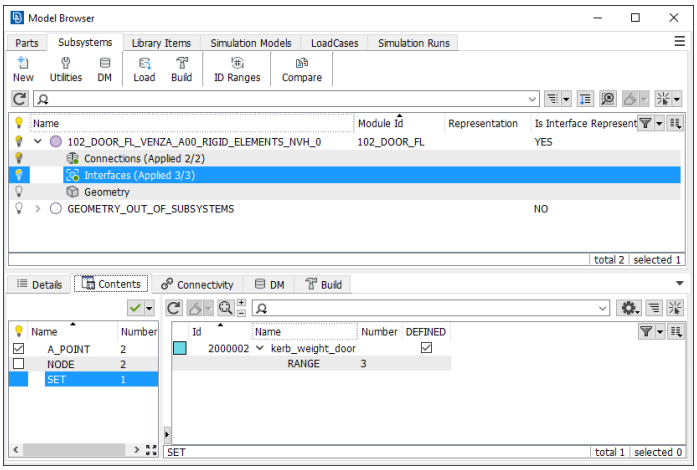
Loading the regular representation of a module through the Model Browser or when downloading it from the DM Browser:

- If the module is of **File Type ANSA**
  Interface sets come along, since they are saved in the ANSA file
- If the module is of **File Type Solver**
  Interface sets may be contained in the solver keyword file or not, depending on the solver. For example in Nastran, the sets are not written out as a keyword in the file. However, in all other decks, sets are written out as a SET or GROUP keyword.

No matter if the interface sets are contained in the solver keyword file, the interface representation ANSA file is automatically merged after the download of the solver keyword file of a module from DM.

This behavior is controlled through the DM setting shown on the left, which is active by default (ANSA.defaults variable: `auto_load_interface_repr_file`)

6.3.4. Association of Interface Sets with Model Containers



In order for a set to be associated with its Model Container, the user will have to drag and drop the set onto the module of interest. Once a set is associated with a module, it will be listed under the Interfaces container in the Model Browser.

If a set is created by a GEB_SB, both the GEB_SB and the set must be dropped into the module.

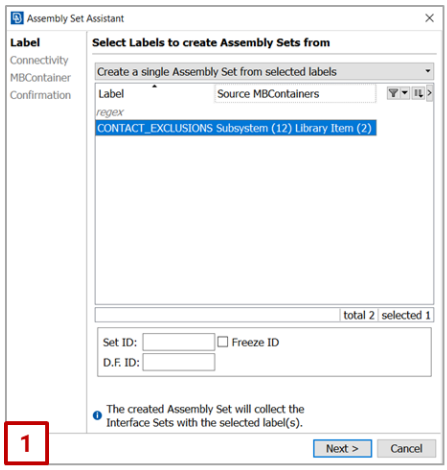
6.3.5. Creating Assembly Sets

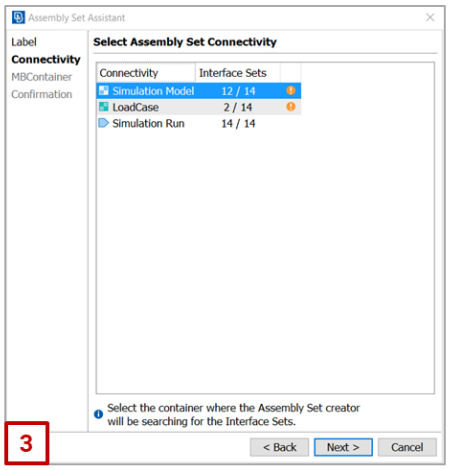
The term Assembly Sets is used for sets that "join" other sets and not exclusively for sets that are used for assembly. In this sense, an Assembly Set may be used for the definition of an initial or boundary condition required for loadcase setup.

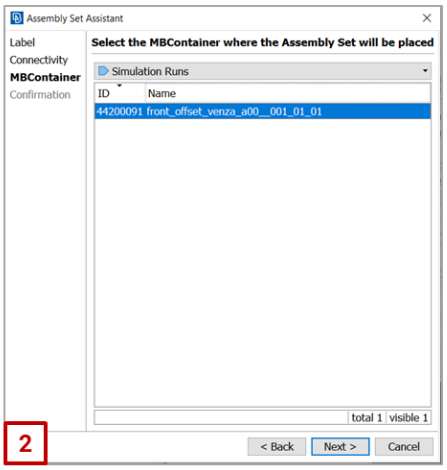
In order to create an Assembly Set the following info must be defined:

- Where to search for Interface Sets: Should the Assembly Set collect sets that reside under the Simulation Model or under the Loadcase?
- Which Interface Sets to collect: The identification of the particular Interface Sets that should be collected under a certain Assembly Set is made through the *labels*
- Where to assign the resulting Assembly Set: Should the Assembly Set be finally assigned to the Simulation Model, to the Loadcase, to a Connecting Subsystem or to a Library Item?

ANSA offers a wizard for the guided creation of Assembly Sets through the option *New>Assembly Set* in the Sets List. An alternative option is to create the Assembly Set directly from the Model Browser window. In particular, the Assembly Set wizard can be invoked by selecting a Connecting Subsystem or a higher level Model Browser container and performing *Actions > Create Assembly Set* through the right-click context menu.







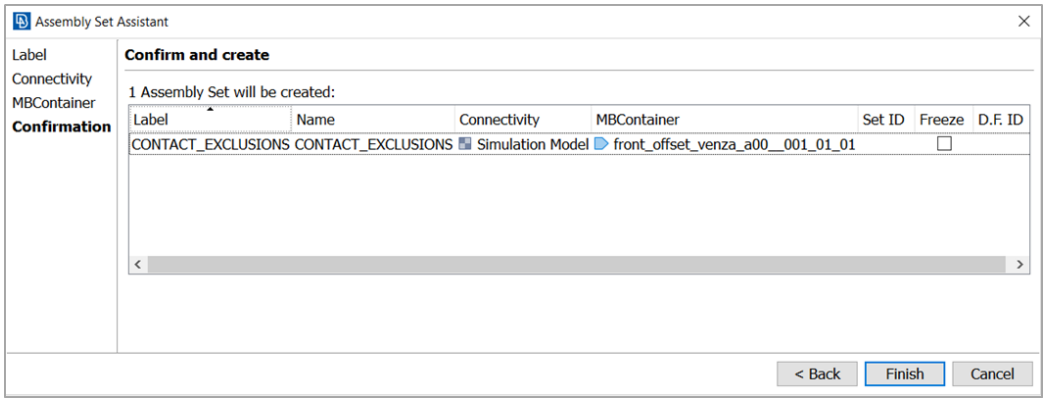
In **page 1**, the user needs to define the labels that will be used for the identification of sets. In case the Interface Representations of the modules have been already loaded and the Interface Sets are available in the ANSA session, all different labels will be listed. In case more than one label is detected and the user selects more than one row in the list, it is possible to control whether all labels must be merged under the same Assembly Set or if one Assembly Set should be created for each label. For the identification of the labels, a regular expression can also be used.

In this page it is also possible to specify the ID of the Assembly Set to be created and also request this id to be frozen.

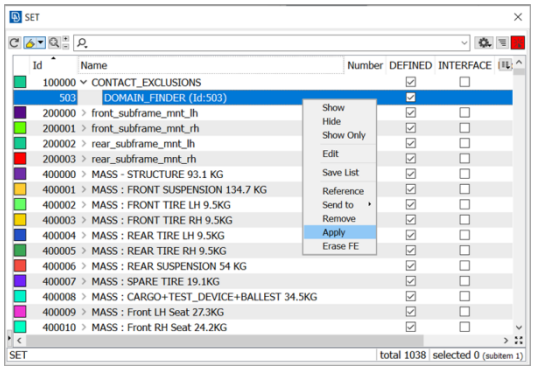
In **page 2**, the user is prompted to define the connectivity of the Assembly Set. This is where the Assembly Set will be searching for Interface Sets. Here the user needs to select an option among Simulation Models, Loadcases and Simulation Runs. Moreover, the number of Interface Sets in each Connectivity group is displayed under the Interface Sets column.

In **page 3**, the user needs to define the container of the Assembly Set. This is where the Assembly Set will be assigned. Here, the user needs to select an option among Simulation Models, Loadcases, Simulation Runs, Connecting Subsystems and Library Items. Then, an entity must be selected from the list.

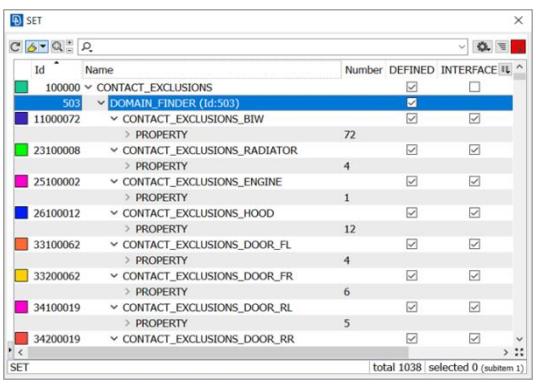
In the final page, a summary of all selected options is given.



On Finish, the Assembly Set is created in the Sets List.



As shown on the left, the Assembly Set contains an entity of type DOMAIN_FINDER. This entity will be used to identify the Interface Sets. The criteria for this identification have been configured by the wizard. The user only needs to **Apply** the Domain Finder in order for the Assembly Set to be populated with the Interface Sets. This action is normally carried out by the Build process.



Finally, the Interface Sets identified by the Domain Finder are listed under the DOMAIN_FINDER entry in the Sets List.

Depending on the user selection in the first page of the wizard, the Domain Finder may search in the Simulation Model or in the Loadcase or in both, by defining the option Simulation Run.

This information is passed to the connectivity of the Domain Finder that controls the search domain. The connectivity can be one of the following values:

- **#SM**: In this case, the Domain Finder will search in the Simulation Model (In the Subsystems and Library Items contained in the Simulation Models and in its Model Setup Entities). In case more than one Simulation Models are available in the session, one of them must be marked as *working* in order for the search to succeed. Otherwise, the search will fail.
- **#LC**: In this case, the Domain Finder will search in the Loadcase (In the Subsystems and Library Items contained in the Loadcase and in its Model Setup Entities). In case more than one Loadcases are available in the session, one of them must be marked as *working* in order for the search to succeed. Otherwise, the search will fail.
- **#SR**: In this case, the Domain Finder will search in the Simulation Run (In the Subsystems and Library Items contained in the Simulation Model and Loadcase and in all Model Setup Entities containers). In case more than one Simulation Runs are available in the session, one of them must be marked as *working* in order for the search to succeed. Otherwise, the search will fail.

The way the Assembly Set will be written out in the solver keyword file depends on the entity that uses it and on the solver. More specifically:

- For Nastran, where no SET keywords are actually written out, the set is expanded to its contents according to the requirements of the entity that uses it. For example, if the set is used by an SPC, then it will be expanded to nodes. If it is used by a PLOAD, it will be expanded to elements. If it is used by a NSM defined on PSHELLs and originally it contained properties, it will be expanded to properties.
- For all other solvers, where a SET keyword exists and can be used for the mass definition of entities, the set doesn't need to get expanded to its contents. However, the type of set that will be written out depends solely on the entity that uses it. For example in Abaqus, if the entity that uses the Assembly Set requires elements, it will be written as an *ELSET. In Ls-Dyna if the entity that uses the Assembly Set can use a set with properties, the set will be written out as a *SET_PART_ADD. In case the Assembly Set is not used at the time of output, the output type can be defined through the OUTPUT TYPE option of the Set card.

6.3.6. Assignment of Assembly Sets to Model Containers

Assembly Sets are assigned to a Model Container at the last step of the Assembly Set Wizard. The user can select as a destination of the Assembly Set any of the following:

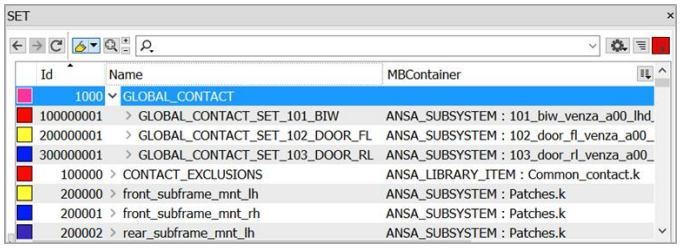
- Simulation Model
- Loadcase
- Simulation Run
- A Connecting Subsystem
- A Library Item

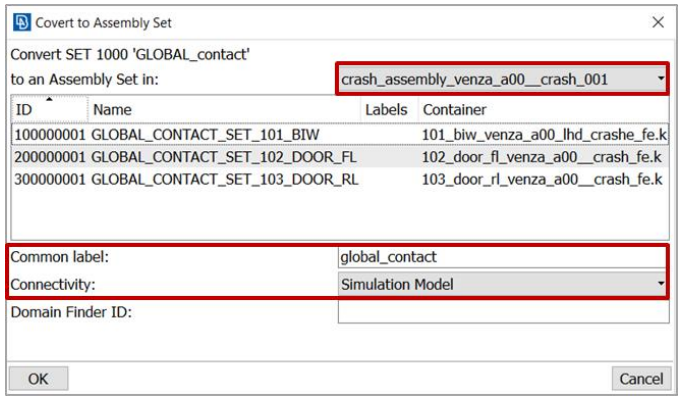
If an Assembly Set is created manually (i.e. without the use of the Wizard), the user must assign the Assembly Set to a Model Container manually. Note that in this case, both the Assembly Set and the Domain Finder must be put in the same container.

6.3.7. Conversion of existing Sets into Assembly and Interface Sets

Legacy models that were not prepared in the Modular Environment, may contain sets that act as “Assembly Sets”, bringing together other sets that belong to different includes. Within the Modular Environment it is possible to convert such sets into actual “dynamic” Assembly Sets. During this process, all “children” sets will be marked as interface sets and a Domain Finder will be created to populate the Assembly Set.

This conversion is activated through the context menu option *To Assembly Set*.

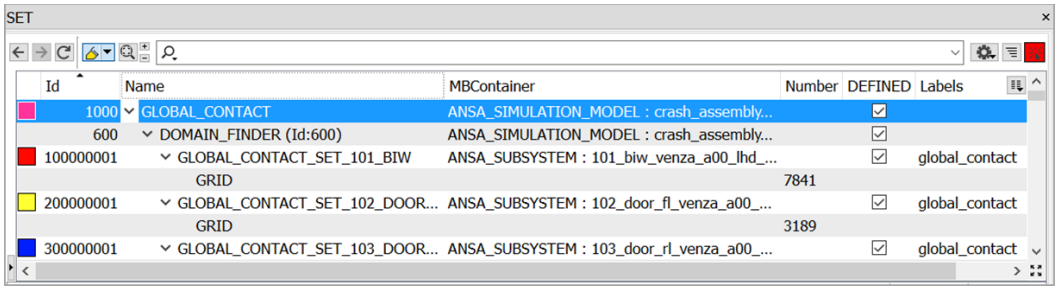




The associated wizard will appear in which the user must define the following information:

- Connectivity.
- Search label of the Interface Sets.
- Container that the Assembly Set will be stored.

Finally, by selecting and applying the generated DOMAIN_FINDER, the Interface Sets which were identified based on the search label are listed under the created Assembly Set on the Sets List.



6.3.8. Composition and use of Assembly Sets - Smart Assembly

The only data required in order to update an Assembly Set is the collection of Interface Sets that reside in the Interface Representation files of Subsystems and Library Items, and the Assembly Set definition that resides in a Connecting Subsystem, in a Library Item or in the Model Setup Entities of the Simulation Model, the Loadcase or the Simulation Run. Thus, we could say that the *Smart Assembly* process introduced in paragraph 6.1.10 is also applicable for the update of Assembly Sets. In this case, the loading of the required data is done in 3 steps:

1. In case the Assembly Set was assigned to a compound Model Container, the Assembly Set with its Domain Finder are loaded through the Definition ANSA File of the compound Model Container
2. The Interface Representation files of all regular Subsystems and Library Items are loaded, through *DM>Load Interface Representation*

3. In case the Assembly Set was assigned to a Connecting Subsystem, the Assembly Set with its Domain Finder is loaded through the standard ANSA file of the Connecting Subsystem

Except for the first step, which is managed manually by the user depending on use-case, steps 2 and 3, with this order, are implemented by the **Load Interfaces** Build Action available on Simulation Models and Loadcases.

After loading, and in order to update the Assembly Set, the Domain Finder must be applied. This task is carried out by the **Realize Internal GEBs** Build Action available on Simulation Models, Loadcases and Simulation Runs.

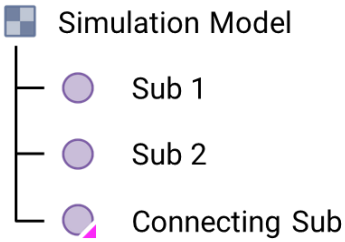
6.4. Organization of intermodular connections in files

Intermodular connections generate entities that tie modules together. In order to control where the entities will finally be written, the user needs to assign the intermodular connections to the appropriate Model Browser Container. There are mainly 3 alternative arrangements for the organization of connecting elements:

1. Put the connecting elements at the same level with the entities they connect
2. Put the connecting elements one level higher than the entities they connect
3. Put the connecting elements within the subsystem they connect

The paragraphs below describe the different set-ups.

6.4.1. Connecting elements at the same level with the entities they connect



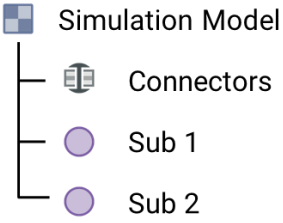
This is the most common organization.

In this arrangement, assuming a connection between Subsystem 1 and Subsystem 2 of the Simulation Model, the connections between Subsystem 1 and Subsystem 2 would be saved in a Connecting Subsystem that is also placed under the Simulation Model, and appears as sibling of Subsystems 1 and 2.

The peculiarity of Connecting Subsystems is that they do not attempt to identify their contents automatically based on connectivity information (remember that Regular Subsystems “pull” their internal connections and Connector Entities automatically, based on their connectivity). Instead, they expect the user to manually drop on them any entity that should be considered an intermodular connecting entity. Thus, a user can assign to a Connecting Subsystem connecting elements, Connection and Connector Entities as well as Assembly Sets with drag and drop.

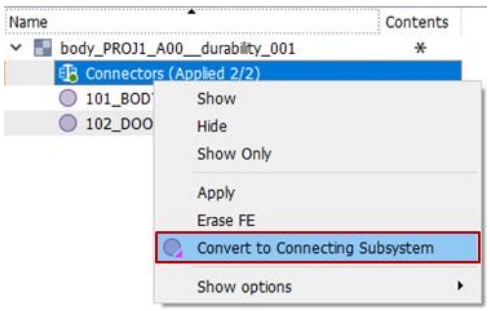
Note that it is possible that not all of the intermodular connections of the model are assigned to a single connecting subsystem. In order to reduce the number of sources that would trigger the creation of a new version of the connecting subsystem, engineering teams usually distribute the intermodular connections in more than one connecting subsystems. This variation is equally valid.

6.4.2. Connecting elements one level higher than the entities they connect



In this arrangement, assuming a connection between Subsystem 1 and Subsystem 2 of the Simulation Model, the connections between Subsystem 1 and Subsystem 2 would be saved as “Connecting” entities of the Simulation Model.

During output of the solver keyword file, a setup like this would lead to the connecting element keywords being written out *in-line* the Simulation Model file.



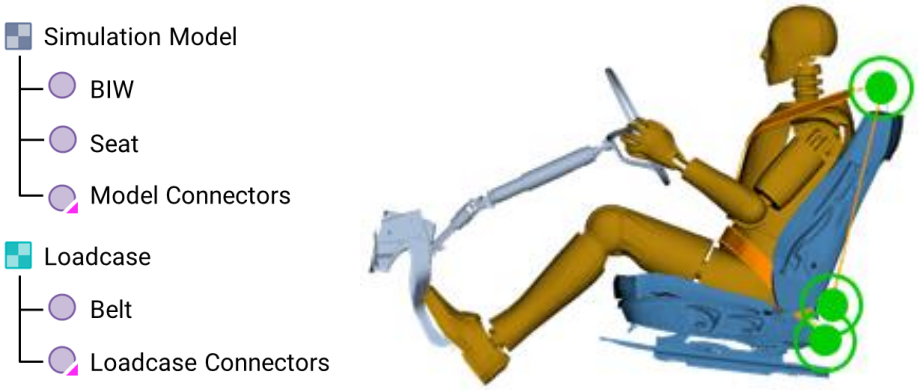
It is possible to move all Connecting entities of a Simulation Model to a Connecting Subsystem that will be assigned to this Simulation Model through the context menu option *Convert to Connecting Subsystem.*

6.4.3. Connecting elements within one of the subsystems they connect

In this arrangement, assuming a connection between Subsystem 1 and Subsystem 2 of the Simulation Model, the connections between Subsystem 1 and Subsystem 2 would be assigned to either of the two. This arrangement is not really supported by the Modular Environment at present.

6.4.4. Connecting elements between the Loadcase and the Model

It is likely that some intermodular connections will need to connect a module coming from the Loadcase with a module coming from the Simulation Model. A typical example is the attachment of a seatbelt on the model. The seatbelt is most probably a Subsystem added to the Loadcase.



In these situations it is advisable to add such Intermodular Connectors at the Loadcase level, since the default order of execution of the Build process facilitates this organization:

Running **Build** at the Simulation Run level, the process is executed bottom-up in the following order:

1. Subsystem
2. Simulation Model
3. Loadcase
4. Simulation Run

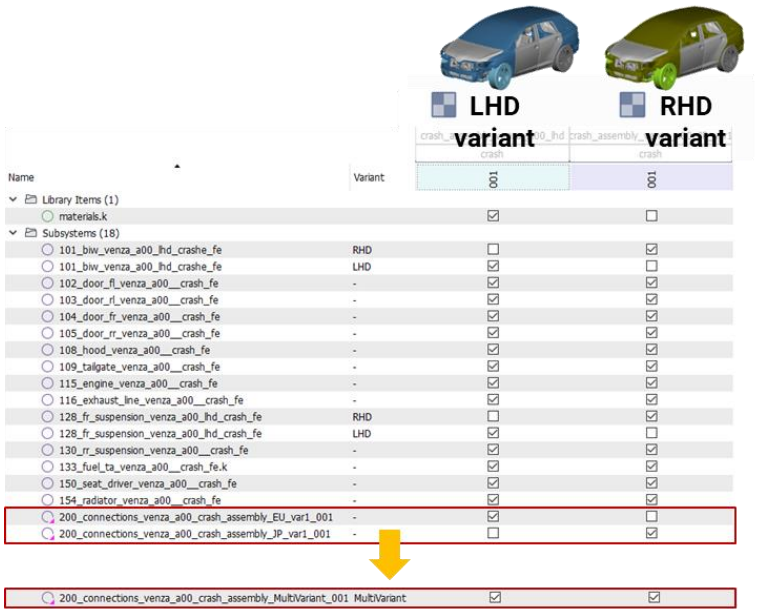
The Smart Assembly actions in this process will be:

- On Simulation Model level:
  - Load Subsystems: It will load the Interface Representations of all Subsystems contained in the Simulation Model and the Intermodular Connectors that connect them
  - Apply Connectors: It will apply the Intermodular Connectors that connect Simulation Model contents
- On Loadcase level:
  - Load Subsystems: It will load the Interface Representations of all Loadcase-specific modules and the Intermodular Connectors that connect these modules with the Simulation Model
  - Apply Connectors: It will apply the Intermodular Connectors that connect the Loadcase to the Simulation Model

6.4.5. Using 150% Connecting Subsystems

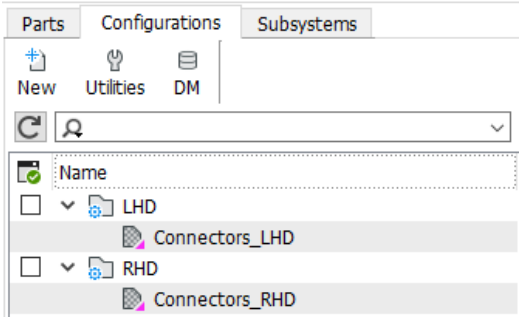
In the usual case where intermodular connections are managed with Connecting Subsystems, it is very likely that Connecting Subsystems will have to be setup in a way that facilitates the model assembly of different Simulation Model variants. For all these cases, 150% Connecting Subsystems can be employed in order to efficiently manage the different variants of Connecting Subsystems.

In a case like the one shown below, where a single Connecting Subsystem is used for all the intermodular connections of the Simulation Model, there will be a big number of commonalities between the intermodular connections of the two Connecting Subsystems. In order to avoid duplication of work and ease the creation and management of intermodular connections, a single Connecting Subsystem can be created to hold the intermodular connections that will be used by both the LHD and the RHD variant of the Simulation Model.



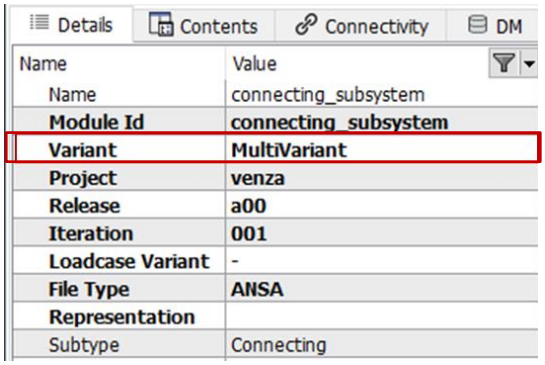
Different Connecting Subsystem for each Simulation Model Variant

Same Connecting Subsystem for both Simulation Model Variants



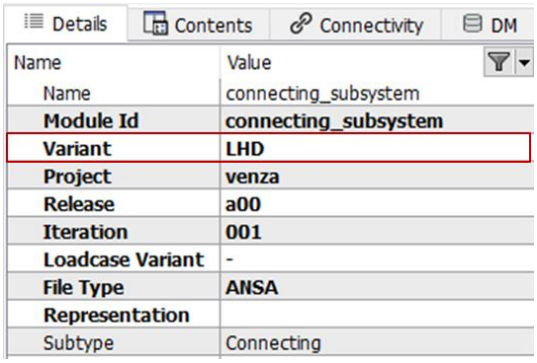
In order to create a Connecting Subsystem that will contain the connections from all model Variants, the required configurations must be created first. The connector entities should be assigned to the respective Parts used in each configuration and then the Parts will be placed in the Subsystem. The result will be a 150% Connecting Subsystem that can be combined with different combinations of regular Subsystems to create the various

Simulation Models.



The 150% subsystem in this case would contain both the *Connectors_LHD* and the *Connectors_RHD* ANSA Parts. The Configuration name is linked with the Variant property of the Subsystem. If no Configuration is active, the Variant of the subsystem is **MultiVariant**.

Once a configuration is activated, the respective variant of the Connecting Subsystem is changed. As a result, the part containing the irrelevant CONNECTOR entities is deactivated, essentially going from the 150% to the 100% model.



During assembly, the correct configuration should be activated, keeping only the connectors required in a particular Simulation Model. This action could be included as a custom action in the Build process of the Simulation Model.

7. Loadcase setup

A Loadcase within the Modular environment is a combination of Library Items and Subsystems. This organization offers several advantages since it allows using instances of a Library Item or Subsystem under the same or different Loadcases, in a slightly different way each time, without actually making copies of the item.

There are three general categories of Library items that can be used with Loadcases:

1. **Plain**
   Simple Library Item which is used under a Loadcase similar to Subsystems.
2. **Loadcase Header**
   Library Item that contains a Loadcase Setup Assistant template. This template can be used to set-up a Loadcase for Nastran or Abaqus solver.
3. **Target Points**
   Library Item that creates a Loadcase for each point in a collection of target points.

Depending on the types of the contained Library items, we have three types of Loadcases:

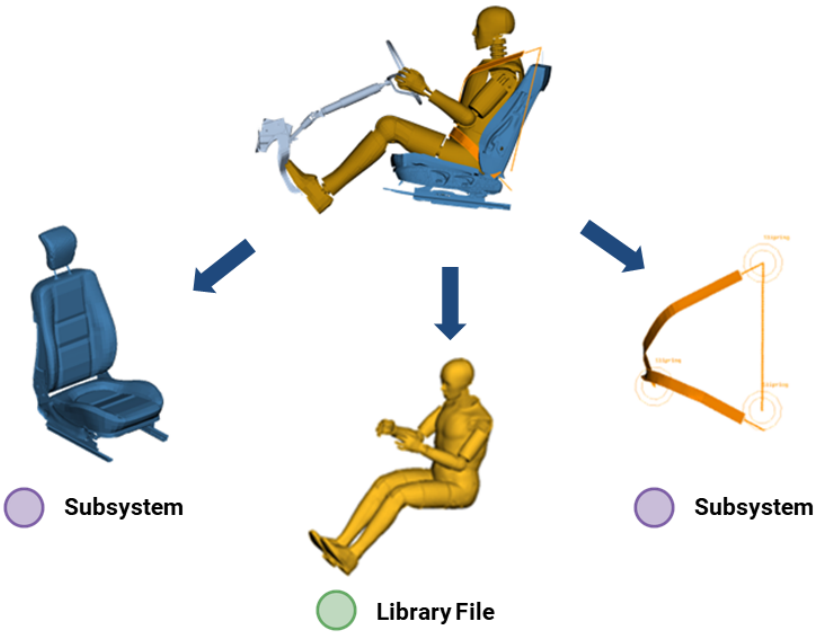
- **Regular**: Contains only plain library items
- **With Loadcase Header**: Contains one Loadcase Header item
- **With Target Points**: Contains one Target Points library item

Note that plain Library items and Subsystems may exist inside a Loadcase with a Loadcase Header or inside a Loadcase with Target Points.

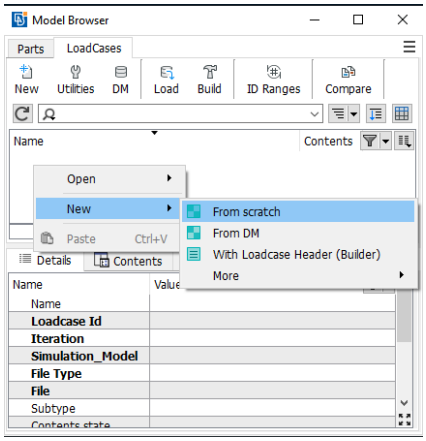
7.1. Creating a Regular Loadcase

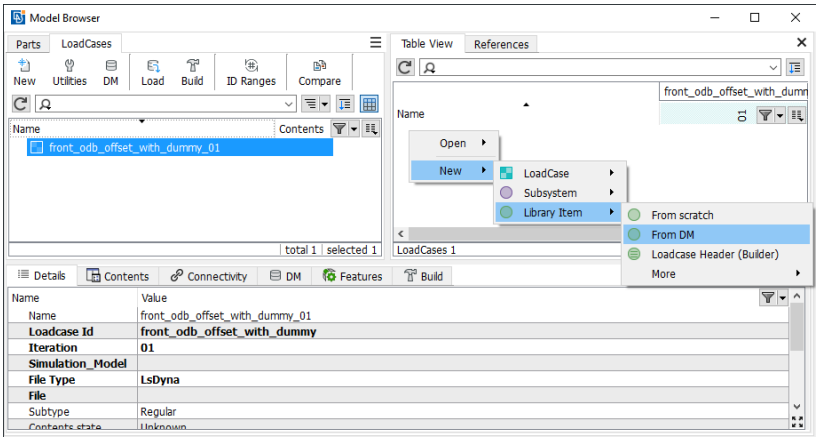
7.1.1. Loadcase Definition

A Loadcase in its simplest format is a collection of plain Library items and Subsystems.

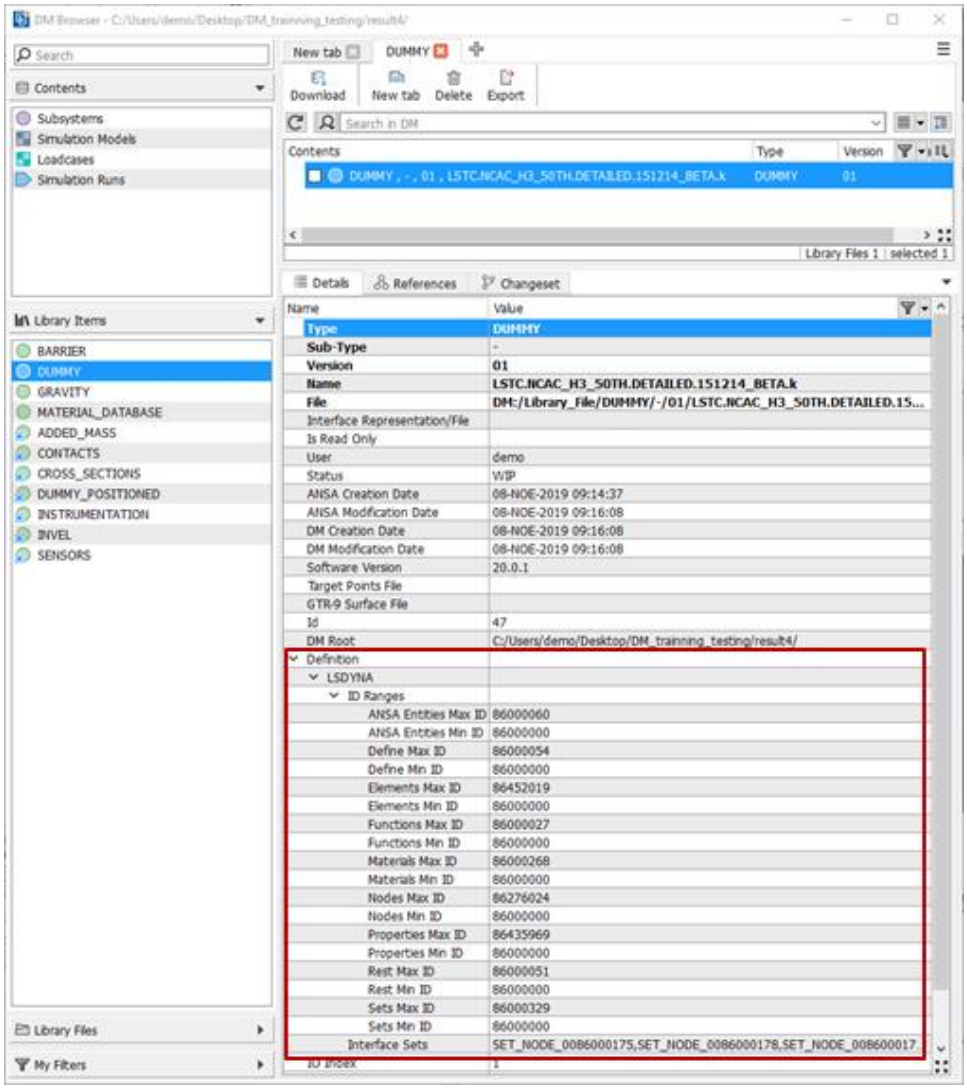


A new Loadcase can be created through the respective option in Model Browser and then it can be populated with contents that are already available in DM:





This process requires that all the Loadcase contents are already available in DM and are indexed. With the term indexing we refer to the process that runs automatically every time a file is added in DM, either with Add or with Save, and extracts information on the contents of the file in order to store it as metadata of the item in DM. The most common indexing metadata are the numbering ranges.



The indexing information is subsequently used during the Adaptation process.

7.1.2. Loadcase Adaptation

The loadcase contents, can be slightly modified with respect to the actual item that is available in DM, in order to participate in a particular Loadcase. The Loadcase adaptation refers to the following actions:

- Handling of numbering ranges
- Transformations and Positioning

The related functionality is described in paragraphs 5.2.1 and 5.2.2.

7.2. Creating a Loadcase with a Loadcase Header

The Loadcase Header is a special library item type, that contains a Loadcase Setup Assistant. The Loadcase Setup Assistant is an ANSA tool that facilitates the generation of the Nastran Header and Abaqus Step Manager. Through the Loadcase Setup Assistant, boundary conditions and output requests can be defined using only the interfaces of a Simulation Model.

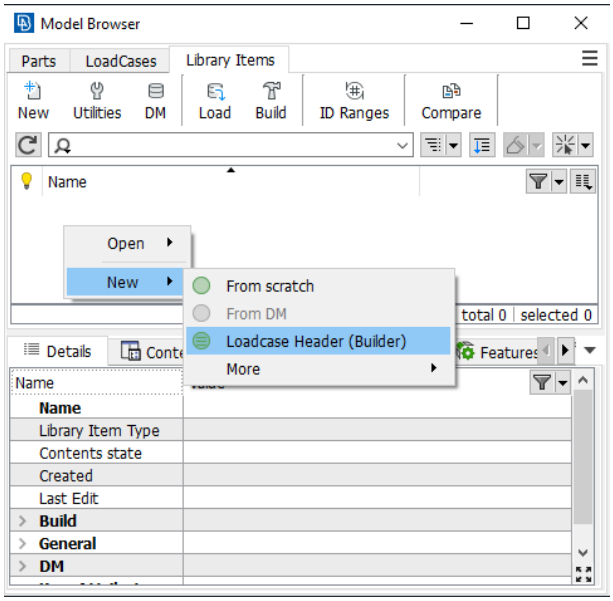
A Loadcase may contain up to one Loadcase Header library item. Such Loadcases are marked with Subtype: "*with Loadcase Header*" to distinguish them from the Regular Loadcases which only contain plain Library Items.

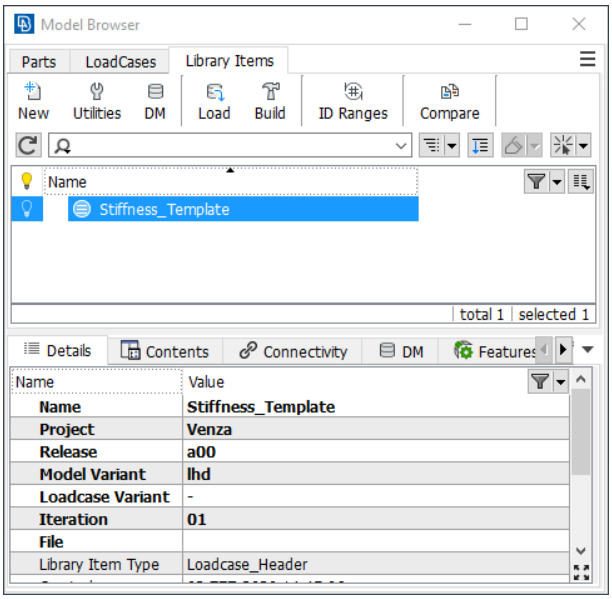
A Loadcase Header can be stored in DM as a template that can be used in different Loadcases and be adapted to different Simulation Models.

The following types of loading and boundary conditions can be defined through the Loadcase Header:

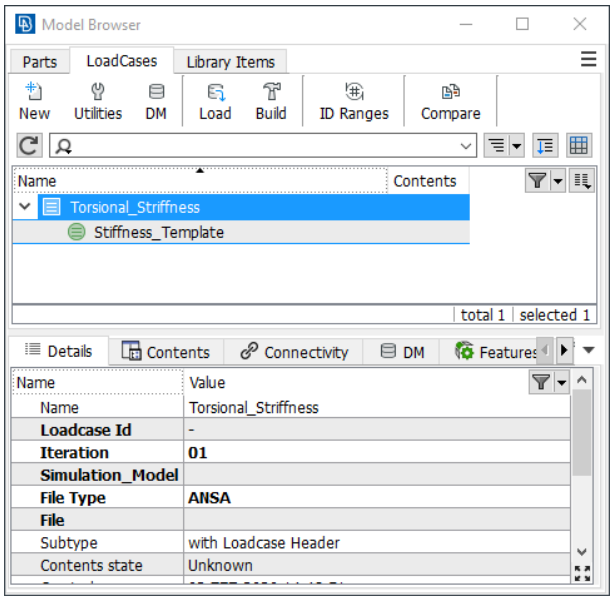
1. Point Boundary Conditions
2. Set-based Boundary Conditions
3. Boundary Conditions defined on the Simulation Model
4. Subcase generator: Transfer function

A new Loadcase Header Library item can be created through the respective option in the Library Items tab.



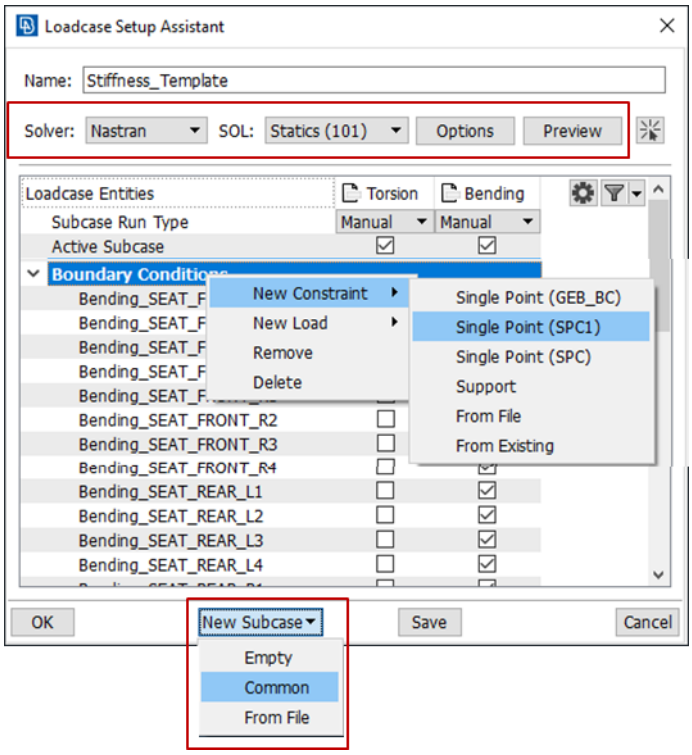


Once created it can be added to a Loadcase, similar to plain Library Items.



Through the Loadcase Header library item it is possible to define:

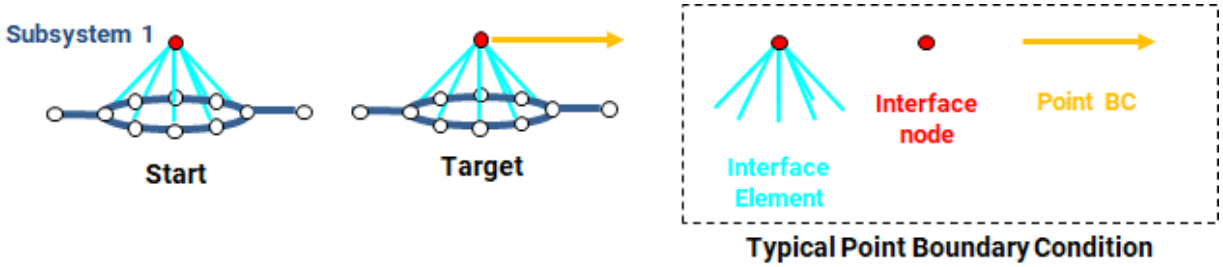
- Solver information
- Subcases
- Boundary Conditions
- Contacts
- Output requests



7.2.1. Point Boundary Conditions

7.2.1.1. Anatomy of a Point Boundary Condition

A typical point Load or Boundary Condition is shown in the image below:

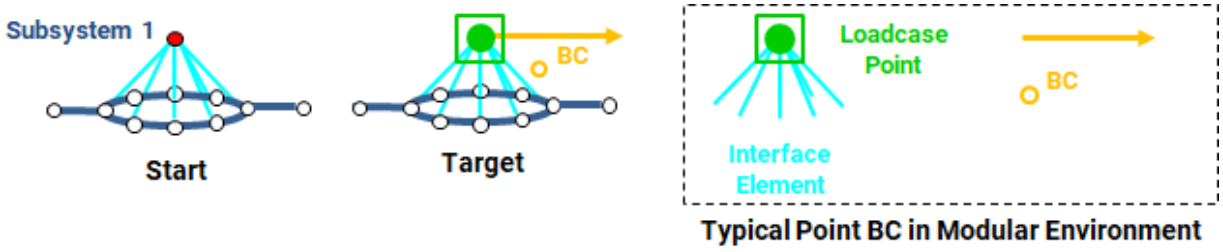


The characteristics of this point Boundary Condition are:

1. A Boundary Condition is applied on an **interface node** that resides in a Subsystem.
2. The interface node is usually connected to geometric features of the Subsystem through an **interface element**. Quite often, an interface node marks a node on the Subsystem geometry, e.g a node of a shell element or end-point of a beam.

In the Modular Run Environment:

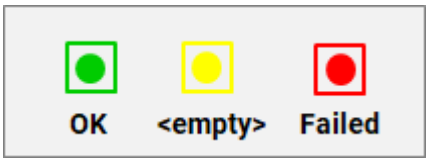
- Interface nodes used for Loadcase setup are marked with **Loadcase Points**.
- The **Loadcase Points** belong to the modules (i.e. Subsystems or Library Items)
- The Boundary Conditions can either be created manually or through a generic entity builder (GEB_BC).
- The Boundary Conditions and the corresponding builders belong to the Loadcase Header.
- The interface elements may pre-exist or be created from the **Loadcase Points** directly.
- The creation of interface nodes and interface elements is carried out on Subsystem level while the creation of the Boundary Conditions is carried out during Loadcase setup.



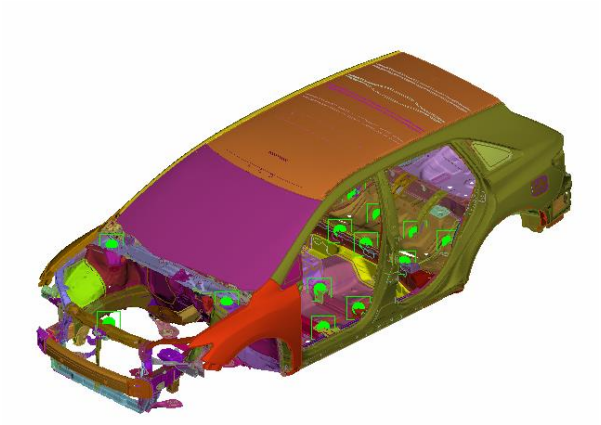
7.2.1.2. Marking interfaces with LC_Points

The Loadcase Points are the entities used to mark the locations where Loads and Boundary Conditions are applied in Subsystems and Library Items. Conventional methodologies mark such interface locations either with specific node ids or specific node names or, in Nastran, with the aid of the field10 node attribute. All three alternatives require a considerable maintenance effort from the model building teams since for each new project, a specification sheet with all interface locations and their proper marking needs to be created and then, for each new version of a module, this specification sheet must be used to mark the interface nodes or verify the marking.

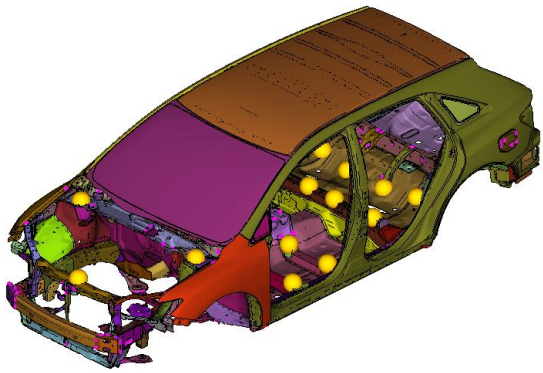
Marking the interfaces with Loadcase Points offers the following benefits:



1. There's a clear visual indication of the interface locations on the model
2. One can see at a glance whether the interface locations are well defined, using the status-based color-coding of Loadcase Points
3. The Loads/Boundary Conditions can be generated by GEB_BC entities that make reference to LC_POINTs. Connectors search for LC_POINTs ***by name and proximity***, relieving the user from the tedious task of monitoring the node ids for all interface locations of assembly.
4. The marked locations can be passed to META with the aid of the alc_aux file, which is a text file with ANSA comments that is written by ANSA on Simulation Run output and describes all the interfaces of a Simulation Run. During the loading of a results file in META, META searches for an alc_aux file with the same name in the same directory and loads it automatically, marking all interface locations.



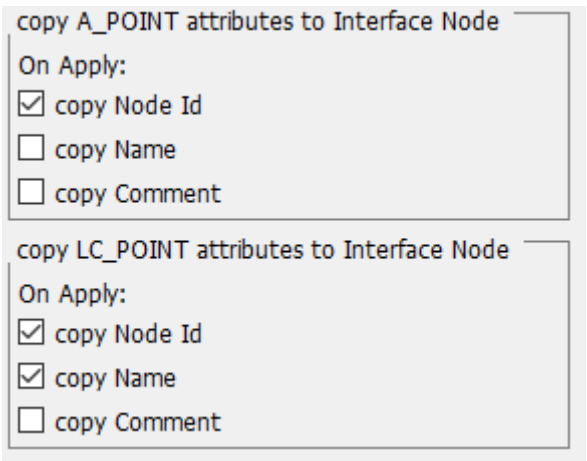
LC_POINTs in ANSA



LC_POINTs in META

Although node names and ids are not required for assembly, loadcase setup or post-processing in the Modular Environment, it is still possible to control the id and name of the interface node through the Loadcase Point, so that models prepared in the Modular Environment are still usable in existing workflows and with 3rd party tools.

This behavior can be activated through Settings>Connections>Connector/Geb/Interface point:



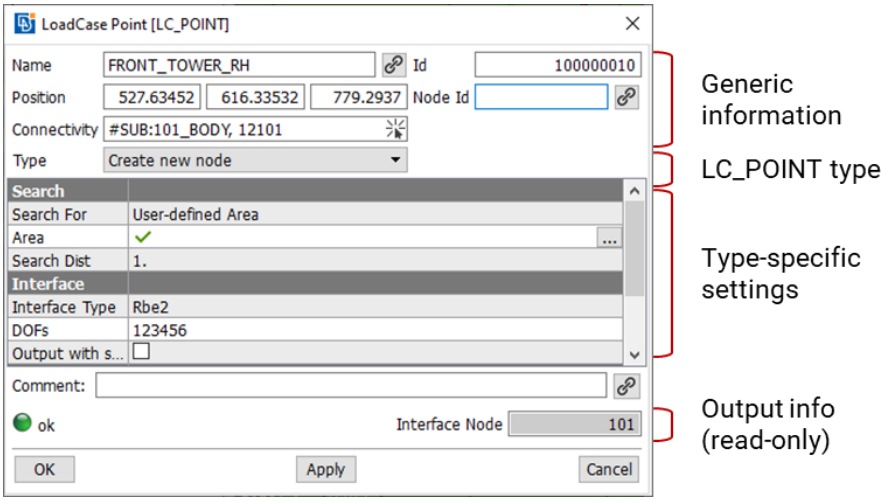
In this form the user can describe the desired behaviour for A_POINTs and LC_POINTs separately. By activating the checkboxes, on LC_POINT Apply, any LC_POINT attributes among the fields Name, Node Id or Comment will be passed to the underlying interface node.

The corresponding settings in the ANSA.defaults are:

- `copy_A_POINT_id_to_node`
- `copy_A_POINT_name_to_node`
- `copy_A_POINT_comment_to_node`
- `copy_LC_POINT_id_to_node`
- `copy_LC_POINT_name_to_node`
- `copy_LC_POINT_comment_to_node`

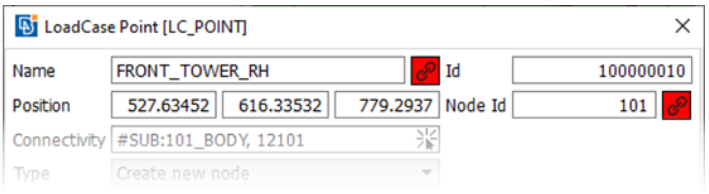
The Loadcase Point card has been redesigned in v21. Its main characteristics are described below:

The top area gathers all generic information, like the LC_POINT name and position, that are used as the main identifiers of the entity, the connectivity, that holds the module id of the base module the LC_POINT belongs to, its Id, that is used as a secondary identifier for compatibility with all other ANSA entities, the Node Id, that can hold the id that should be “pushed” to the identified Interface Node on “Apply”.



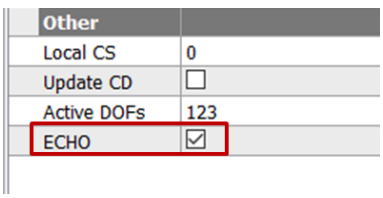
Right below there is the LC_POINT type, which determines the settings that will be used for the identification and marking of the interface node. Description of the supported LC_POINT types is given in the paragraphs that follow.

Finally, in the bottom area, except for the Comment field, the Status and the Interface Node of the LC_POINT are given, both being read-only results of the LC_POINT realization.



The toggle buttons with the chain icon next to the Name, Node Id and Comment fields indicate whether the respective LC_POINT attribute must be “pushed” to the identified Interface Node on “Apply”.

Their default state is controlled from the ANSA.defaults settings described previously and can be toggled individually, in order to change the “copy to node” behavior of particular attributes for a particular LC_POINT.



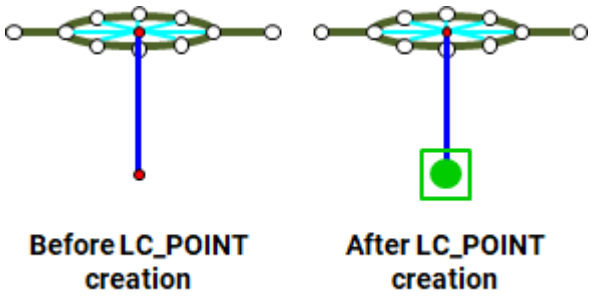
In case of Nastran deck, it is possible to selectively echo the Interface Node of a Loadcase Point. Activating the ECHO option in the LC_POINT card will add the ECHOON/ECHOOFF keywords before and after the respective GRID in the Nastran keyword file.

Note: The fields as shown in the new card do not necessarily correspond to the names of columns in the Database Browser lists and may not be usable for the “getter” and “setter” script functions. In order to get the Database-

Browser-compatible card field names, one can still access the old layout of the LC_POINT card by holding down the shift key while editing an LC_POINT (i.e. shift + double click in the Database Browser list).

There are two main set-ups for Loacase Points that cover the vast majority of interface entity marking use-cases:

Interface Point Type 1: Mark existing node



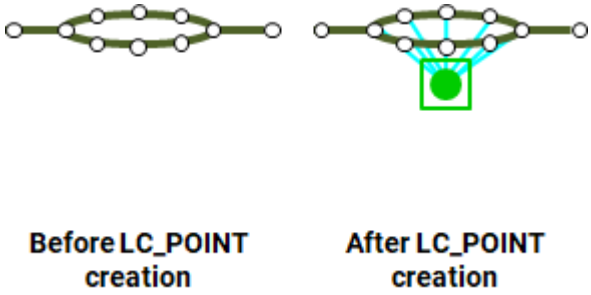
In this case, the user can create an LC_POINT that marks an existing node of the model. Such LC_POINTs can be easily created through the function LC_POINT > New [Node], where the user is prompted to select one or more nodes that will be marked by Loadcase Points.

In this case, the LC_POINT connectivity is auto-filled with the Module Id of the Subsystem the node belongs to (or the Library Item name)

The key settings of these LC_POINTs are:

- **Connectivity**: The Subsystem or Library Item the node belongs to
- **Search**: Identify node by reference, by name or by proximity

Interface Point Type 2: Create new node



In this case, the user can create an LC_POINT at a certain location and search for some geometric features whose nodes will be finally connected with a branch element (RBE2 or RBE3). Finally, the interface node will be a new node that will be created as the central node of the branch entity at the LC_POINT location. Such LC_POINTs can be easily created through the function LC_POINT > New [Location], LC_POINT > New [On Holes (Auto)] and LC_POINT > New [On COG].

The key settings of these LC_POINTs are:

- **Connectivity**: The Subsystem or Library Item where the searched geometric features belong
- **Search**: User-defined area or through Interface Sets
- **Interface**: RBE2, RBE3 or rigid patch

7.2.1.3. Interfaces information in DM

When a Module (Subsystem or Library Item) is saved in DM, the Assembly and Loadcase Points and their FE representations are stored in a separate file named '*interface_representation.ansa*' that is stored as an attached file of the module in DM.

This functionality is similar for Loadcase Points and Assembly Points and it is described in paragraph 6.1.4.

7.2.1.4. Loading interfaces information from DM

Once the interface information has been created in each Subsystem or Library Item and has been saved in DM, it is possible to load in ANSA the interface information alone, in order to set up the Loadcase. Low level, this is achieved with the function *DM>Load Interface Representation*.

This functionality is similar for Loadcase and Assembly points and it is described in paragraph 6.1.5.

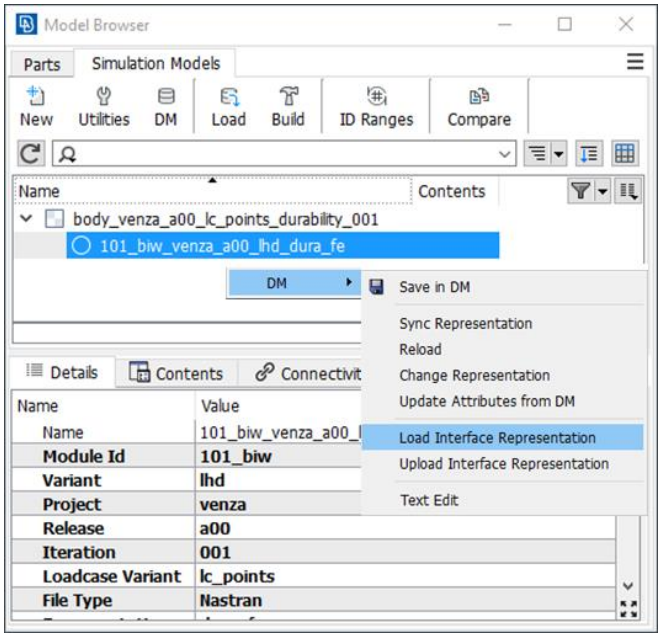
7.2.1.5. Association of interfaces with Model Containers

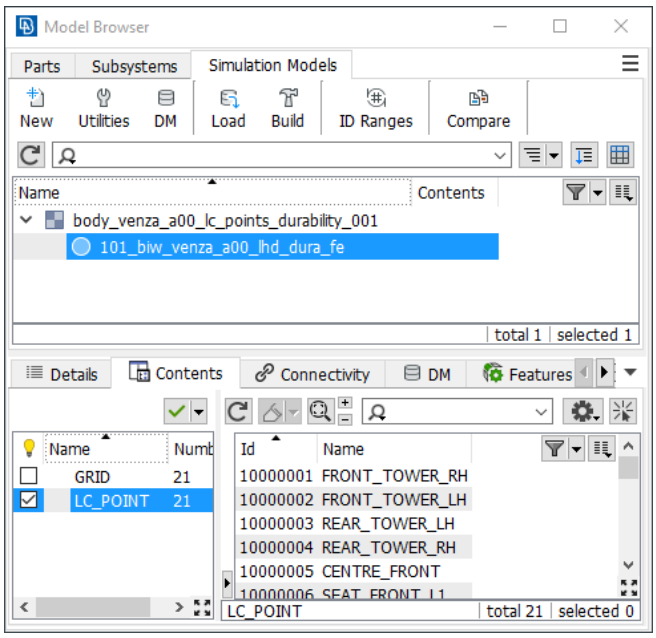
The parent module of Assembly and Loadcase Points is assessed automatically based on their connectivity.

This functionality is similar for Loadcase Points and Assembly Points and it is described in paragraph 6.1.8.

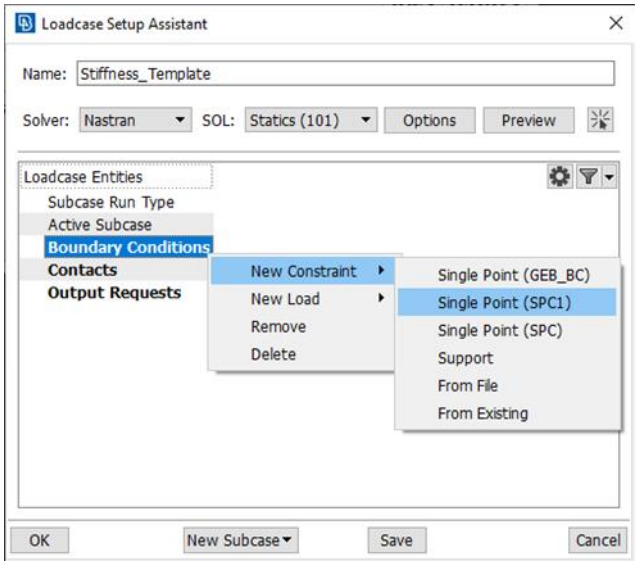
7.2.1.6. Creation of Point Boundary Conditions

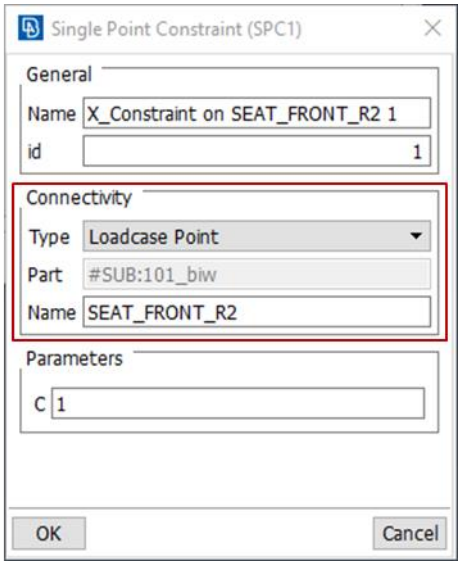
Point Loads and Boundary Conditions can be created based on the interfaces of Modules (Subsystems or Library items) that are available in DM. The Interface representation of a Module is loaded from DM by selecting the *DM>Load Interface Representation* context menu option.





Through the Loadcase Header, Boundary Conditions and Output Requests can be defined on the loaded interface points:

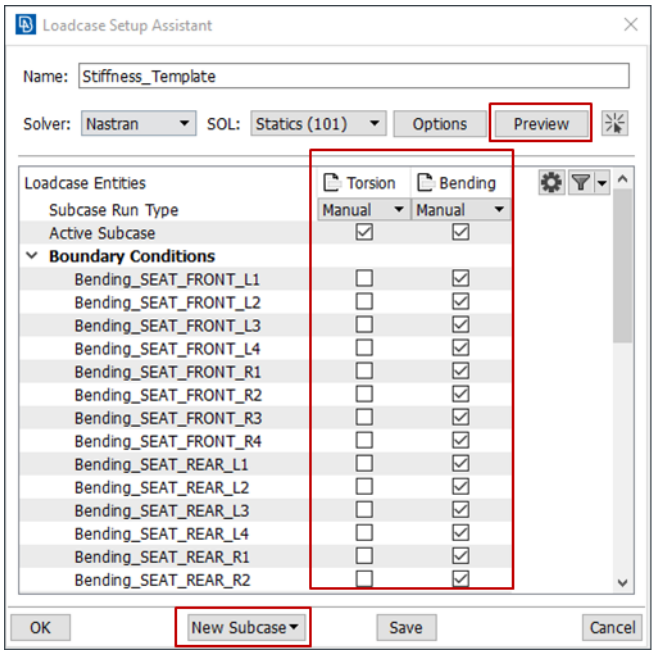


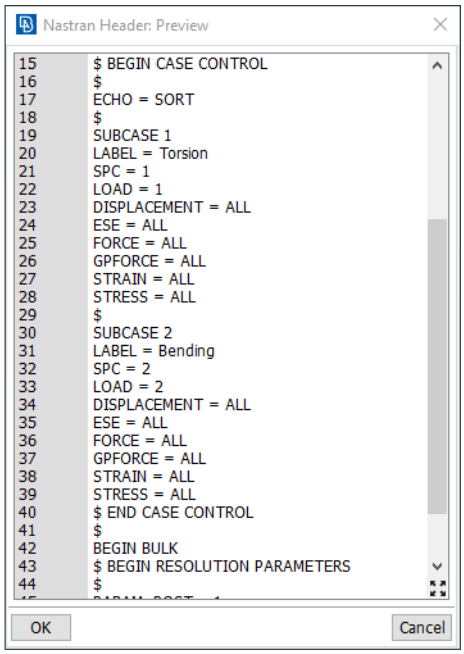


By selecting a Loadcase or Assembly point, the connectivity fields of the selected Boundary Conditions are automatically filled. Note that the Loadcase or Assembly point are mentioned within the Loadcase Header by Name.

Instead of setting up the loadcase using the interfaces of an existing Simulation Model, it is possible to use undefined interface entities. However, this requires that the names of the interface entities are known beforehand. Through the Loadcase Header, it is possible to define a new Boundary Condition by specifying directly the Name and the Type of an interface point, instead of selecting an existing entity. ANSA creates an undefined interface point to apply the specified Boundary Condition. During the Build of the Simulation Run, the undefined interface points of the Loadcase Header are merged by name with the defined interfaces of the Simulation Model (see paragraph 7.2.1.7).

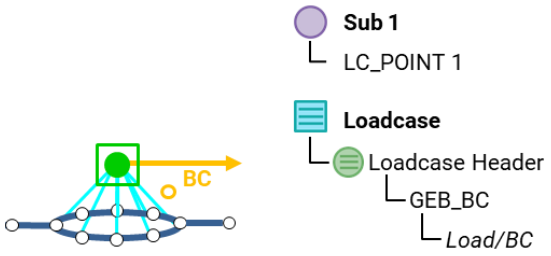
New Subcases can be created through the Loadcase Header and the contents (Boundary Conditions and Output Requests) of each Subcase can be defined. Finally a preview of the generated header can be displayed.



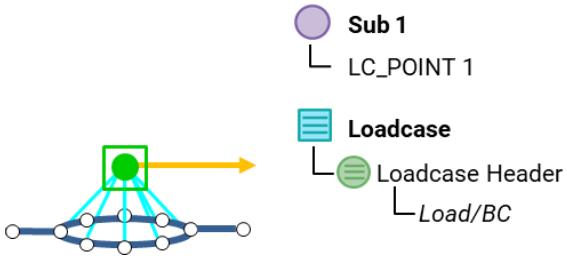


7.2.1.7. Decomposition/Composition

Once the interface location of a point Load/Boundary Condition is marked with a Loadcase Point, it is possible for ANSA to break down the Load/BC during modular save (decomposition) and reassemble it automatically on Load (composition). In the sketches below we see two typical examples of point Loads/BCs.



Case 1: BC generated by a GEB_BC



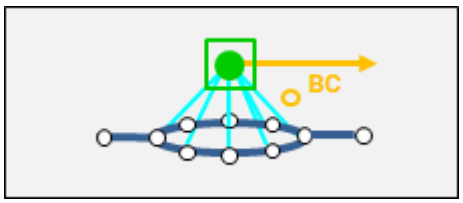
Case 2: BC created manually

The procedures described below cover both cases.

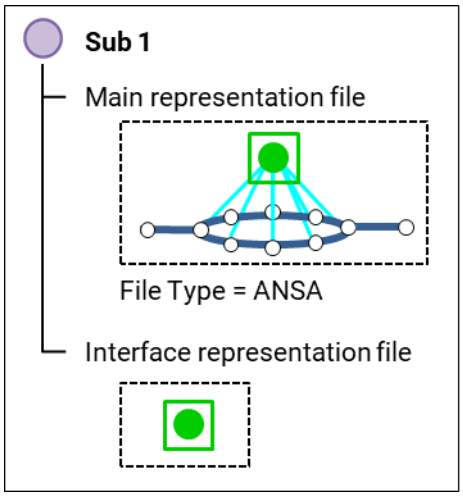
Decomposition during modular Save

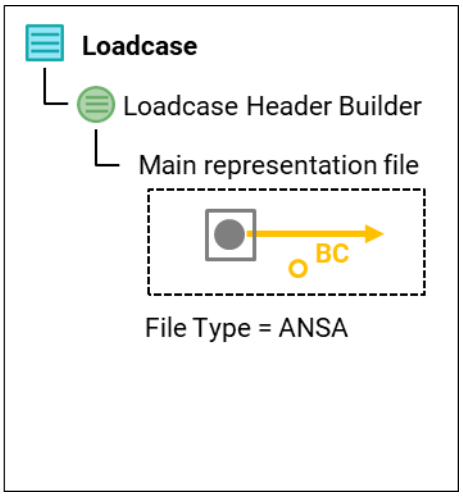
Given that the interface points have been marked with Loadcase Points, during Save of the Loadcase Header, ANSA detects that the Loadcase Points do not belong to the Loadcase Header, and generates automatically *undefined* Loadcase Points in the saved file. These undefined Loadcase Points mark the nodes at the application points of the Loads/BCs, indicating that these nodes *should not really exist in the Loadcase file*. At the same time, the nodes are marked as auxiliary (AUXILIARY=YES) so that they wouldn't be written out in a solver keyword file in case the Loadcase is output and, at the same time, indicating that during id conflicts with other nodes, they should be offset in all cases.

Before modular "Save"



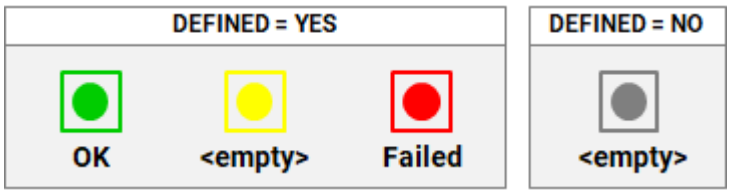
After decomposition





The existence of undefined Loadcase Points allows the user to open the Loadcase Header and make modifications in the point Loads/BCs without having to care about the node ids at their application points. As long as the application points are marked with undefined Loadcase Points, the Loadcase can be rebuilt at any time.

Note

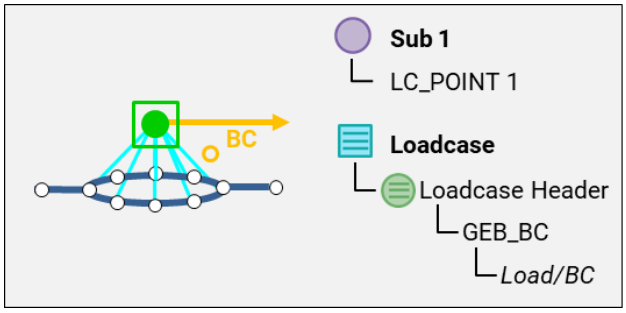


Undefined Loadcase Points have grey color and empty status at all times.

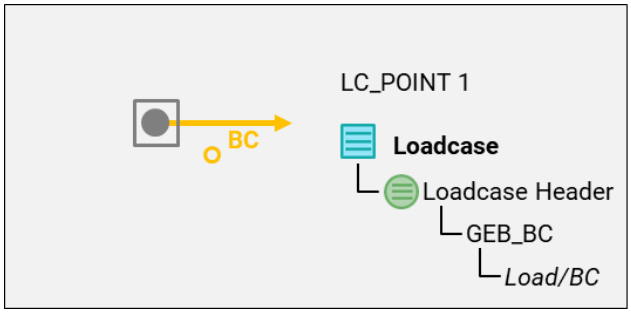
Decomposition during Unload

Upon unloading a module (i.e. Subsystem or Library Item) that contains defined interface points used by entities that belong to a different module, ANSA automatically creates undefined interface points. These undefined interface points enable the automatic re-assembly of the affected entities when the module is loaded back.

Before "Unload"



After "Unload"



Composition during Load

The reassembly of the Loadcase that takes place during "Load" is based on the inherent capability of Loadcase Points to get merged under certain circumstances. Two Loadcase Points get merged when:

- The one Loadcase Point is defined and the other is undefined
- The two Loadcase Points have the same name or have the same name and position, based on the value of the ANSA.defaults variable ***merge_LC_points*** which can get the values ***equal name*** and ***equal name and position.***

During the merging of an undefined with a defined Loadcase Point:

- The Node Id marked by the undefined Loadcase Point is replaced by the Node Id marked by the defined Loadcase Point
- If during this process, indeed, the Node Id marked by the undefined Loadcase Point gets updated, the Loadcase Header is marked as *modified* (magenta), indicating that a new version of the Loadcase must be created
- Otherwise, if the Node Id marked by the undefined Loadcase Point remains the same, the Loadcase Header is not marked as *modified* (magenta), indicating that the existing version of the Loadcase can still be used.

With this functionality, the only data required in order to update the Loadcase file is the collection of *defined* Loadcase Points that reside in the Interface Representation files of Subsystems and Library Items, and the *undefined* Loadcase Points, that reside in the Loadcase Header. Thus, we could say that the Loadcase setup process in ANSA is a 2-step process where:

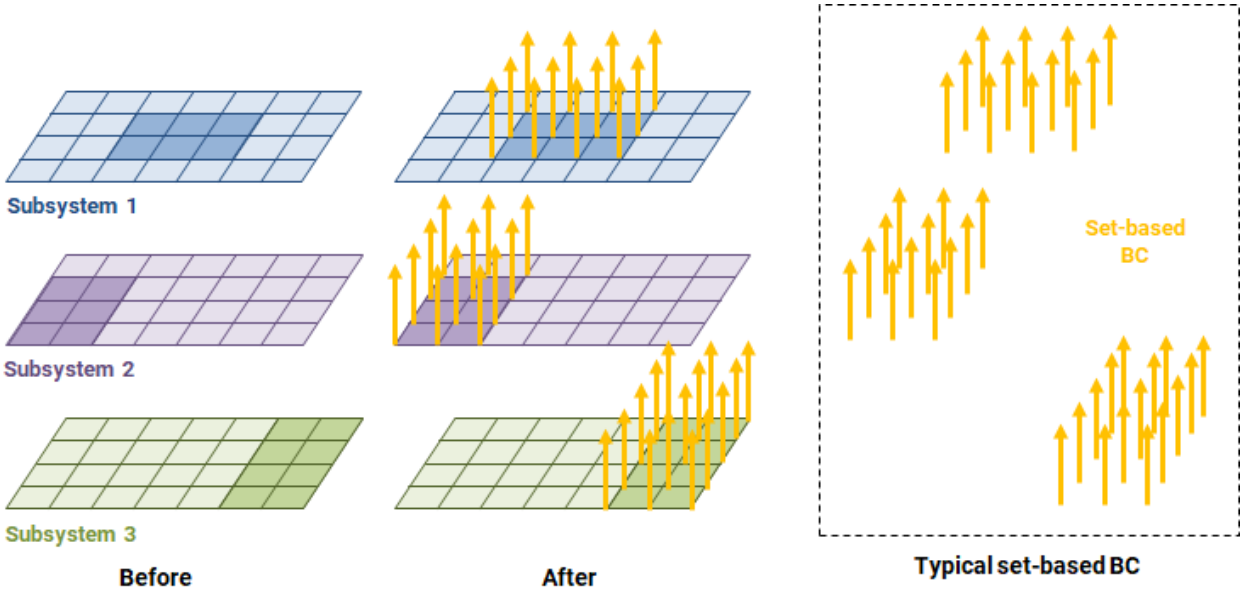
1. The Interface Representation files of all regular Subsystems are loaded, through *DM>Load Interface Representation*
2. The standard ANSA file of the Loadcase Header is loaded, through *Load*

After composition, and in order to verify the Loadcase assembly, the user should check that there are no remaining Loadcase Points with "undefined" status. This is definitely an indication of error. Additionally, the user could check that there are no "orphan" defined Loadcase Points. An "orphan" Loadcase Point may be an indication of a potential problem, but, in most of the cases, it's a normal situation.

7.2.2. Set-based Boundary Conditions

7.2.2.1. Anatomy of a set-based Boundary Condition

A typical set-based Load or Boundary Condition is shown in the image below:

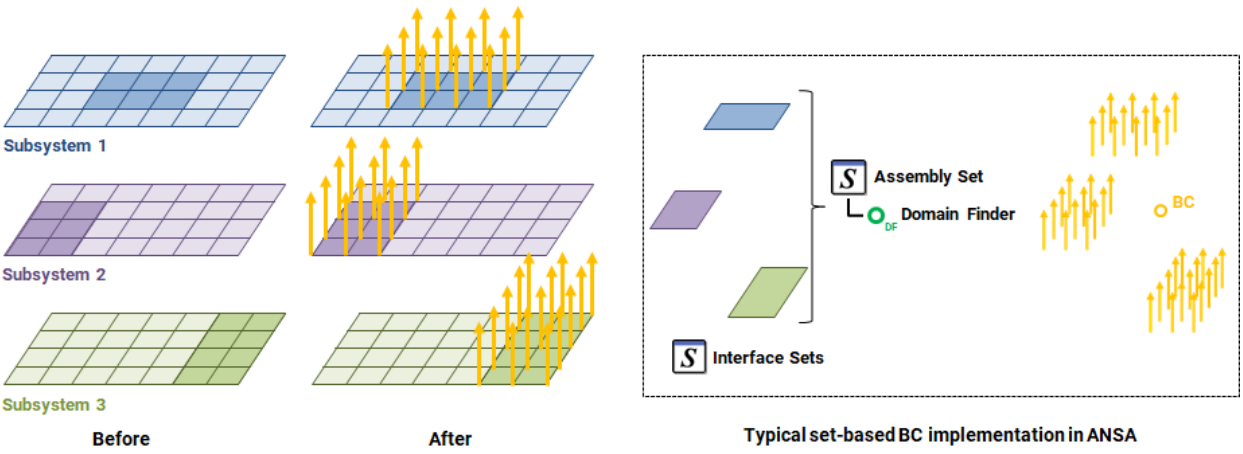


The characteristics of a set-based Boundary Condition are:

1. Each module contributes nodes, elements or properties by adding them to a **labeled interface set** that resides in the module
2. In order for a set-based Boundary Condition to make reference to a single set that contains all the entities contributed by each module, an **Assembly Set** is created, that collects all the subsets and is finally used by the entity.

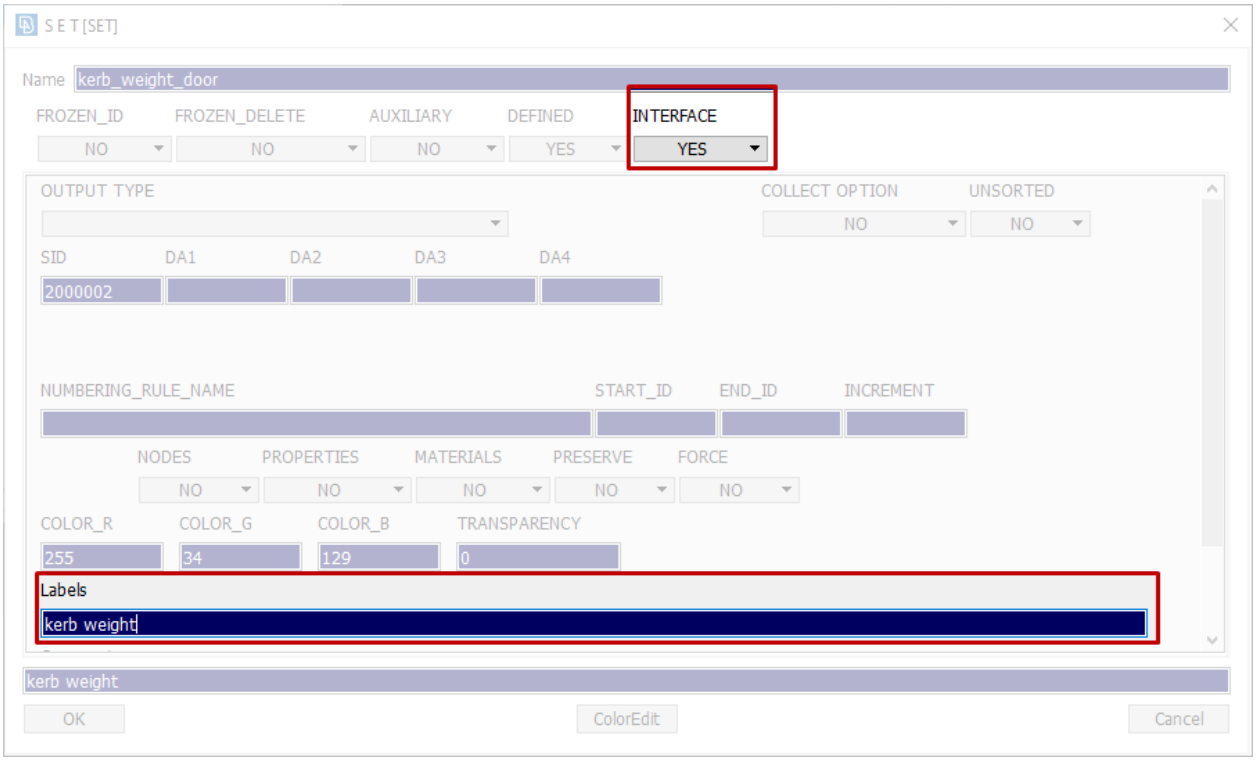
In the Modular Run Environment:

- Interface sets are standard sets marked with the flag **INTERFACE=YES** and labeled with one or more labels that are defined comma-separated in the field **labels**
- The interface sets belong to the modules (i.e. Subsystems or Library Items)
- A Boundary Condition may be defined on a single Interface Set or on an Assembly Set.
- Assembly sets are standard sets that contain a **Domain Finder**, which is an ANSA entity used to search for other entities, in this case interface sets. Whatever entities are found from the Domain Finder are automatically considered contents of the Assembly Set.
- The Boundary Condition can either be created manually or through a generic entity builder (GEB_BC).
- The generated Loads, BCs and the corresponding builders belong to the Loadcase Header.
- The creation of interface sets is carried out on module level while the creation of the Assembly Set is carried out on assembly level.

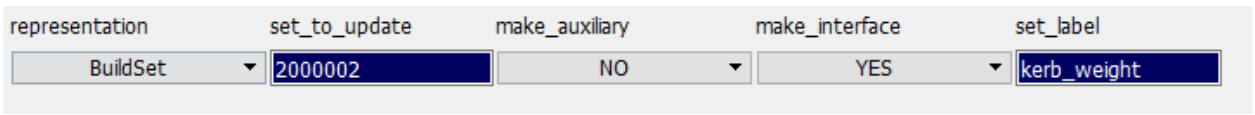


7.2.2.2. Marking interface Sets

Any set can be marked as an *interface* set by activating the INTERFACE flag in the set card.



Alternatively, interface sets can be created with a Generic Entity Builder (GEB_SB) by activating the **make_interface** flag, and, optionally, setting a label in the **representation** section of the GEB_SB card.



7.2.2.3. Interface Sets information in DM

When a Module (Subsystem or Library Item) is saved in DM, the Interface Sets are stored in a separate file named '*interface_representation.ansa*' that is stored as an attached file of the module in DM.

This functionality is described in paragraph 6.3.3.

7.2.2.4. Loading interface Sets information from DM

Once the interface information has been created in each Subsystem or Library Item and has been saved in DM, it is possible to load in ANSA the interface information alone, in order to create assembly sets or use the interface sets directly during Loadcase setup.

This functionality is described in paragraph 6.3.4.

7.2.2.5. Association of Interface and Assembly Sets with Model Containers

The parent module of an Interface Set is assessed automatically based on the Set contents.

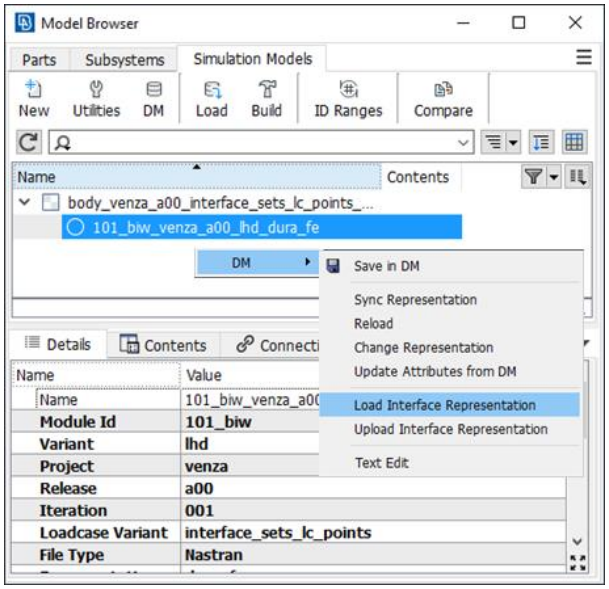
Assembly Sets are assigned to a Model Container at the last step of the Assembly Set Wizard. This functionality is described in paragraph 6.3.7.

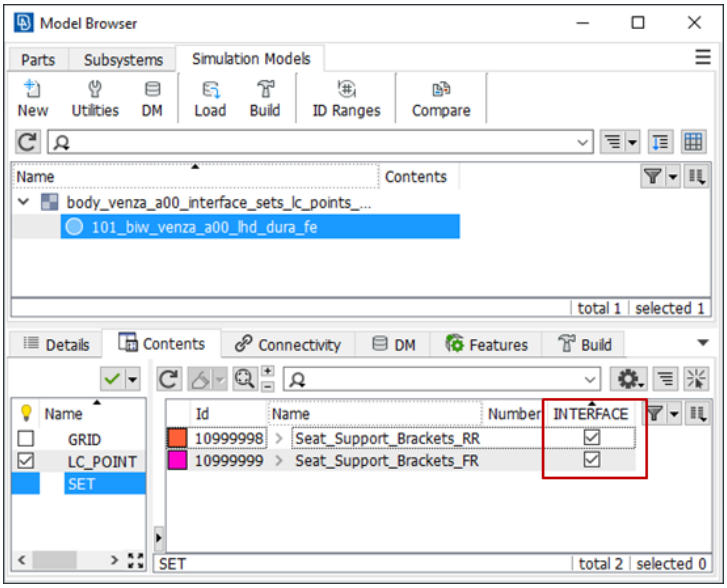
7.2.2.6. Creation of Assembly Sets

The term Assembly Sets is used for sets that “join” other sets and not exclusively for sets that are used for assembly. The creation of Assembly sets is described in paragraph 6.3.6.

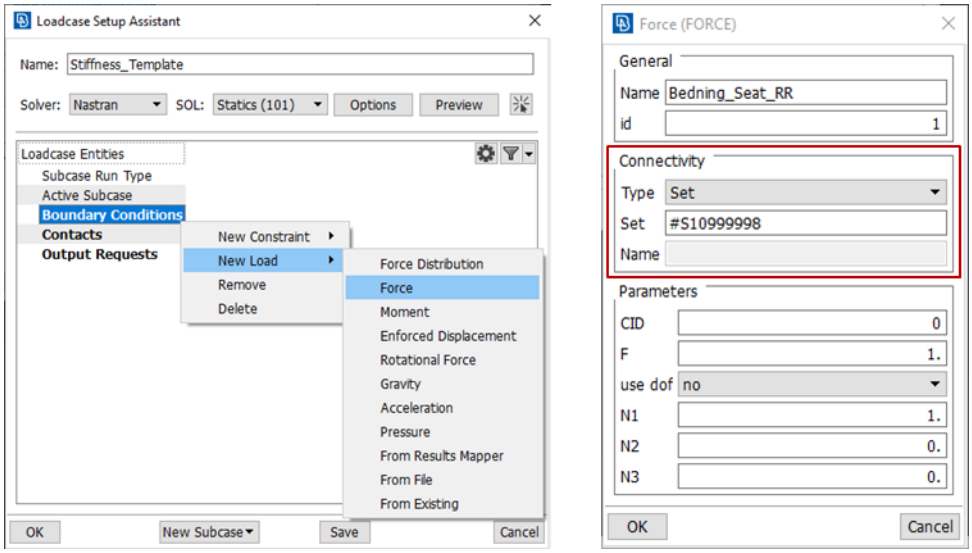
7.2.2.7. Creation of set-based Boundary Conditions

Set-based Loads and Boundary Conditions can be created using the interfaces of Modules that are available in DM. The Interface representation of a Module is loaded from DM by selecting the *DM>Load Interface Representation* context menu option.





In case that the interfaces of the Modules contain Sets, it is possible to define Boundary Conditions, Contacts and Output Requests using these interface sets:



Similarly, instead of using interface Sets, Assembly Sets created with a Domain Finder entity, can be used.

7.2.2.8. Decomposition/Composition

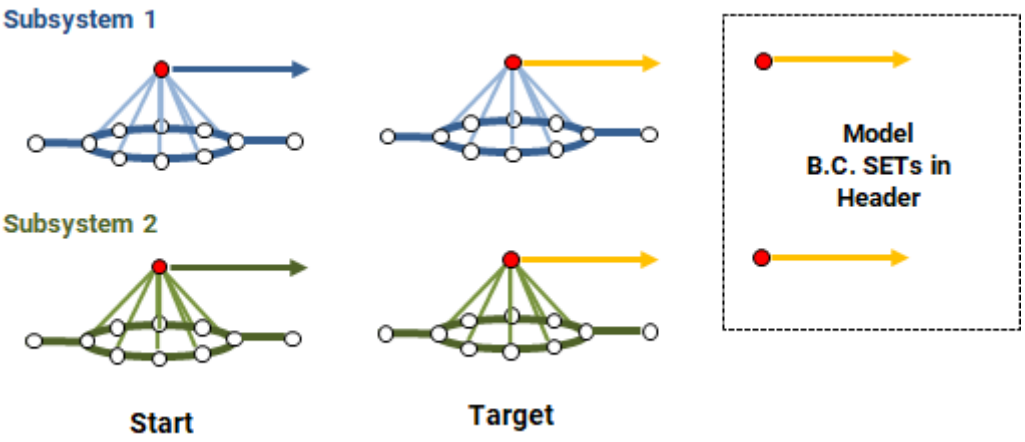
The decomposition and composition in case of set-based Boundary Conditions, defined directly on Interfaces Sets, is similar to the functionality described for Interface Points in paragraph 7.2.1.7.

Regarding Boundary Condition applied on Assembly Sets please refer to paragraph 6.3.8.

7.2.3. Boundary Conditions defined on Simulation Model

7.2.3.1. Anatomy of Boundary Conditions defined on Simulation Model

In case of Nastran, it is possible to define Boundary Conditions (i.e. SPC/MPC/NSM/DMIG/MFLUID) on Module level. During the Loadcase setup such items should be identified and activated in the generated Header.

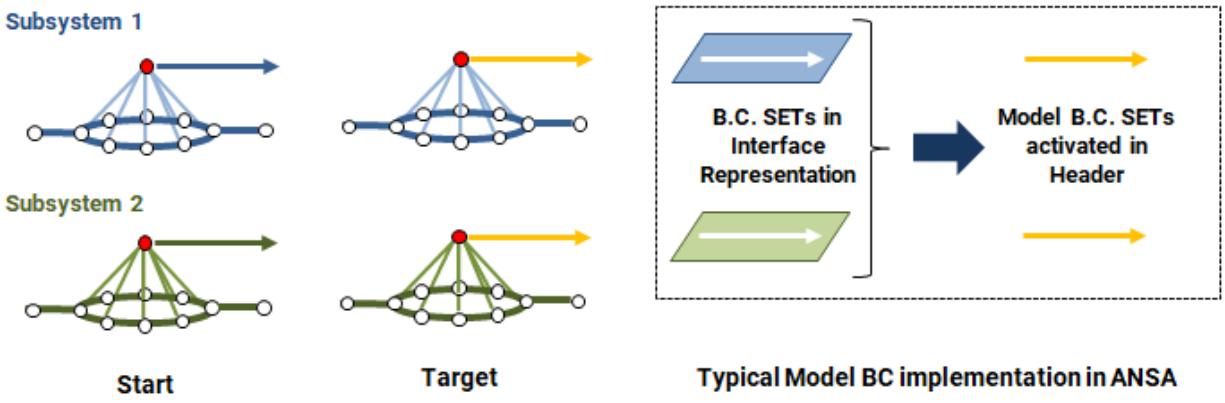


The characteristics of this case are:

1. A Boundary Condition is applied on an interface node or set of a Module.
2. The Boundary Condition entity is placed inside the Module.
3. The Boundary Condition is activated in the Header.

In the Modular Run Environment:

- The B.C. SETs that correspond to a Module's Boundary Conditions are included in the interface representation.
- In case of B.C. SETs used by several Modules, only one Module should contain the actual B.C. SET ("DEFINED: YES") while the rest should contain copies marked as "DEFINED: NO".
- The Loadcase Header automatically detects the Model B.C. SETs and creates the corresponding reference within the Header.
- Model B.C. SETs may be included in the common Subcase or in all Subcases.



7.2.3.2. B.C. SETs information in DM

B.C. SETs that correspond to BCs defined on Module level are considered part of the Module's Interface representation. They are automatically stored in a separate file named '*interface_representation.ansa*' that is stored as an attached file of the module in DM.

This functionality is described in paragraph 6.3.3.

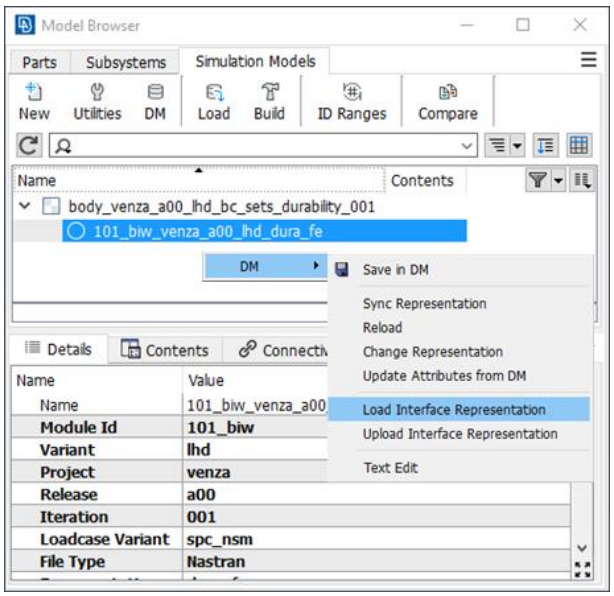
7.2.3.3. Loading B.C. SETs information from DM

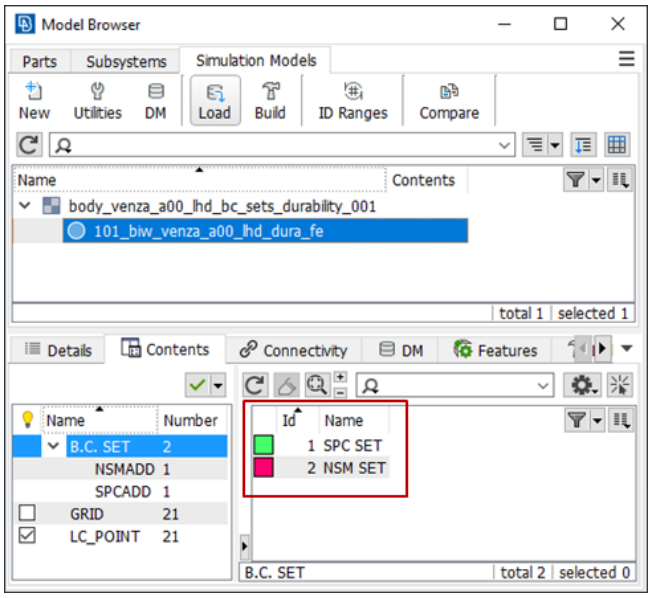
Once the interface information has been created in each Subsystem or Library Item and has been saved in DM, it is possible to load in ANSA the interface information alone, in order to use the B.C. SETs directly during Loadcase setup.

This functionality is described in paragraph 6.3.4.

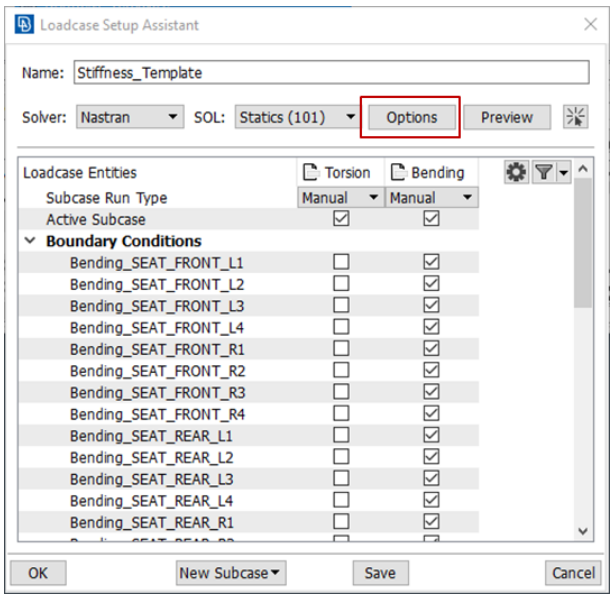
7.2.3.4. Activation of Model Loads/BCs in Loadcase

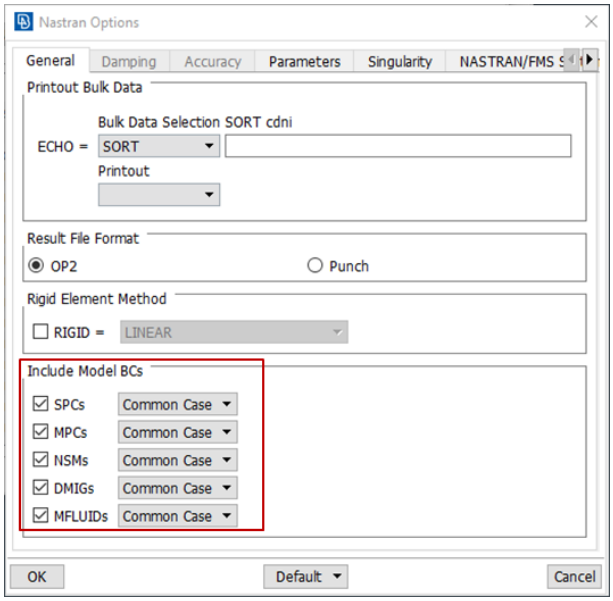
Any B.C. Sets present in a Module's Interface representation, are loaded from DM by selecting the *DM>Load Interface Representation* context menu option.





Through the Loadcase Header it is possible to control centrally whether all the Model Boundary Conditions of a specific type will be referenced in the common Subcase or in all Subcases:

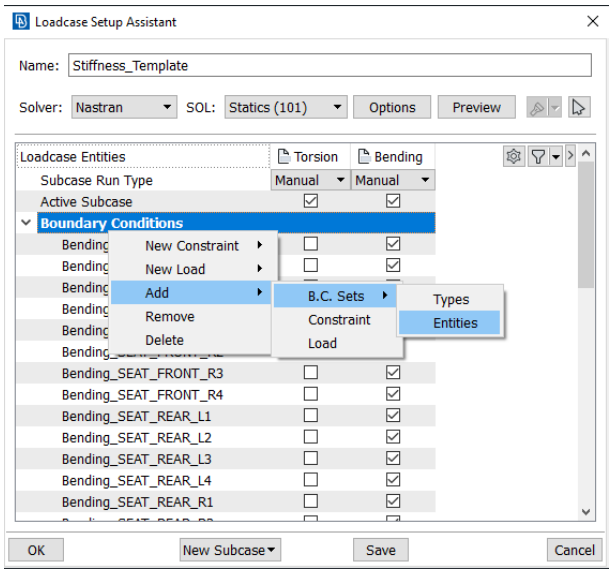


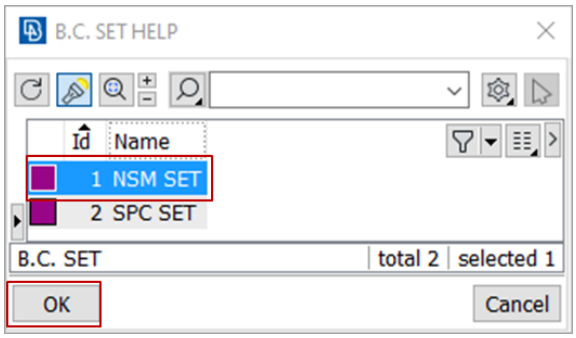


During Save in DM or Output the Model Boundary Conditions are included in the Nastran Header.

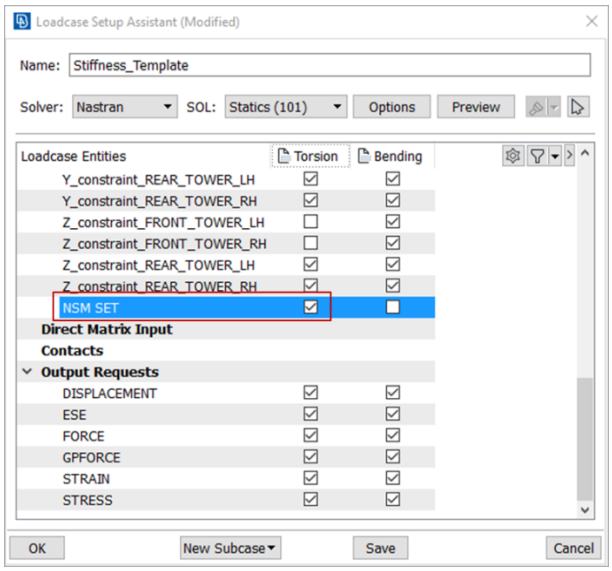
```
$ BEGIN CASE CONTROL
$
NSM = 2
SPC = 1
ECHO = SORT
$
SUBCASE 1
LABEL = Torsion
SPC = 14
LOAD = 1
DISPLACEMENT = ALL
```

Apart from this, it is possible to manually select certain Model Boundary Conditions:





And activate them only under specific Subcases:

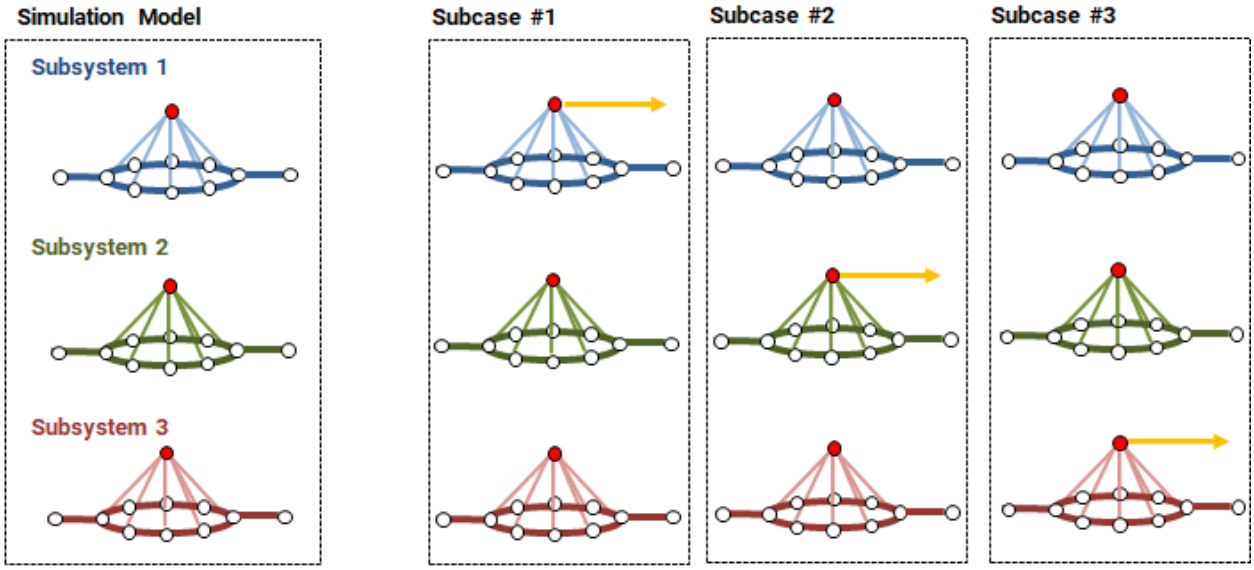



```
$ BEGIN CASE CONTROL
$
ECHO = SORT
LABEL = common
$
SUBCASE 1
LABEL = Torsion
SPC = 14
LOAD = 1
NSM = 1
DISPLACEMENT = ALL
$
SUBCASE 2
LABEL = Bending
LOAD = 2
SPC = 2
DISPLACEMENT = ALL
$ END CASE CONTROL
```

In case of manual activation of Model Boundary Conditions, any centrally defined options for the specific type are overridden.

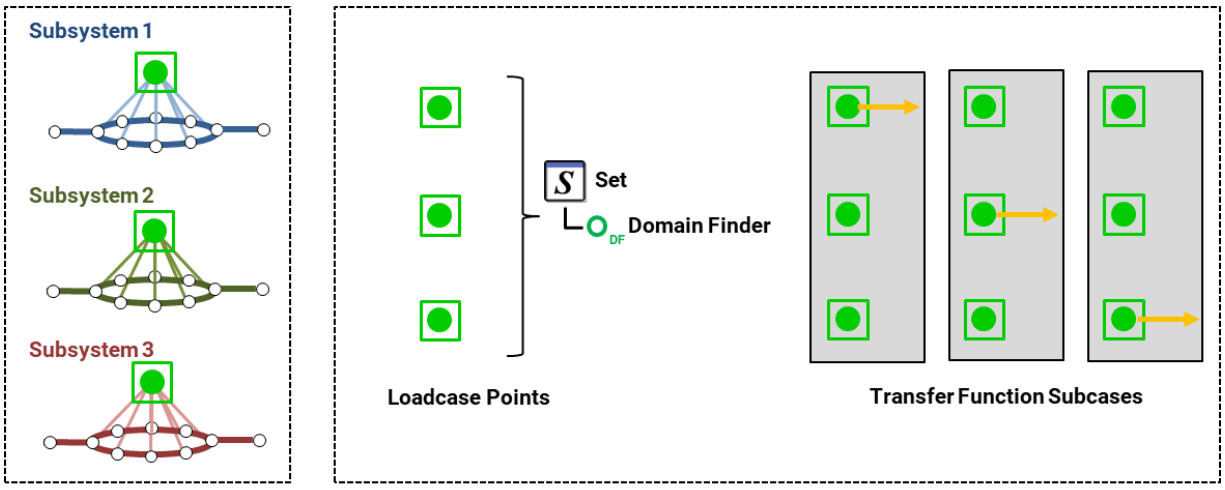
7.2.4. Transfer Function

The Loadcase Header offers the possibility to set up different Subcases by loading separately each degree of freedom for all the Loadcase Points that are placed in selected interface Sets. This functionality is activated by marking a Subcase as 'Transfer Function'. The use of the Transfer Function is mainly applicable to Modal Frequency Response analysis.



In the Modular Run Environment:

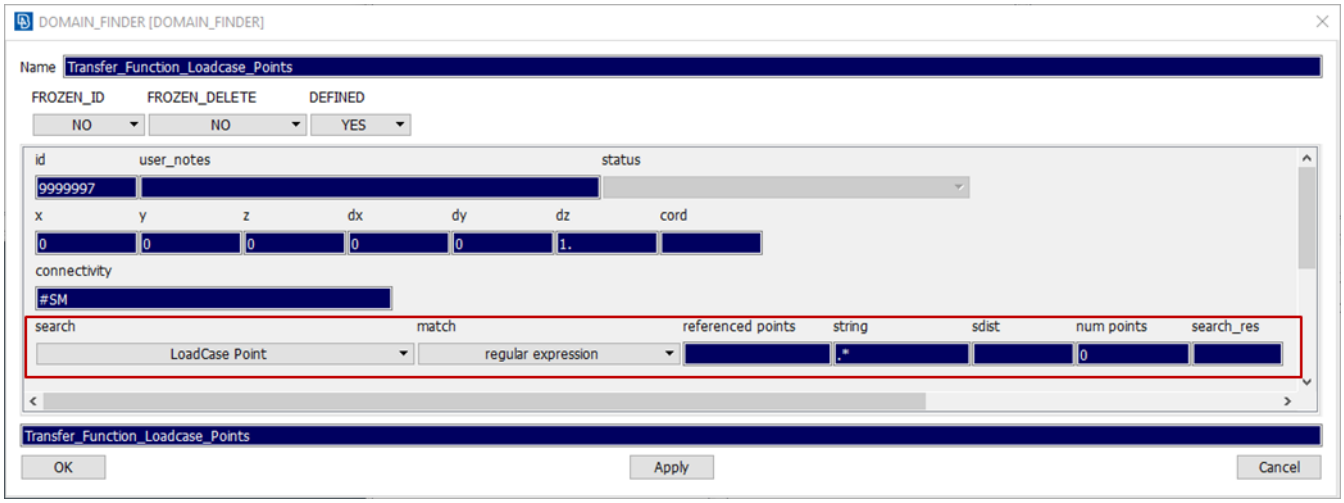
- The Loadcase Points belong to the modules (i.e. Subsystems or Library Items).
- The Loadcase Points of the Model are collected in a Set with the aid of a Domain Finder.
- The creation of the Loadcase Points is carried out on module level while the collection of the Loadcase Points under a single Set is carried out on Loadcase level.
- All generated Subcases share the same Boundary Conditions, apart from Loads.



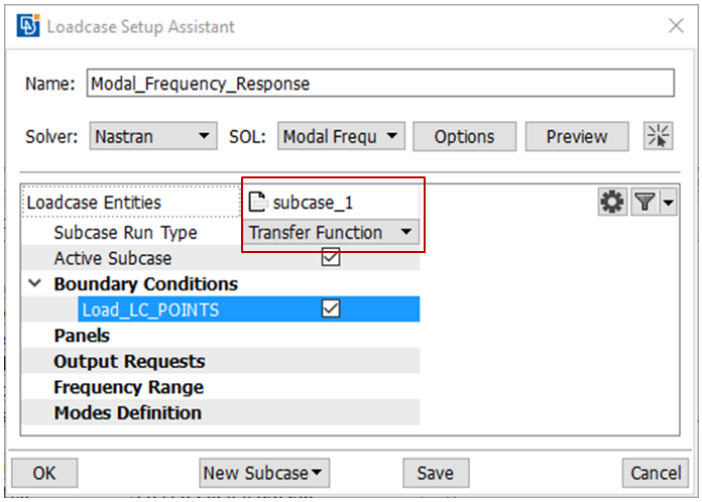
The first steps for the generation of a Transfer Function Subcase are:

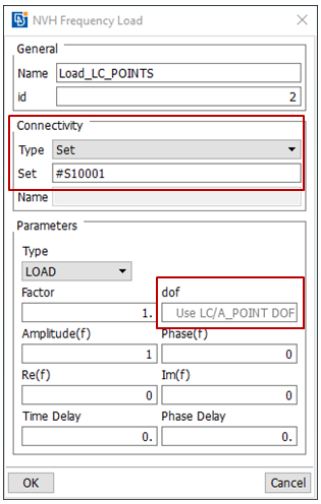
- Load the Interface representation of Subsystems from DM (see paragraph 6.3.4)
- Collect all the Loadcase Points using a Domain Finder.
- Add the Domain Finder in a Set

An example of the Domain Finder set up is shown below:



The Loading conditions of a Transfer Function Subcase that will be expanded are defined on the Set which contains the Domain Finder and must have the '*Use LC/A_POINT DOF*' option active.

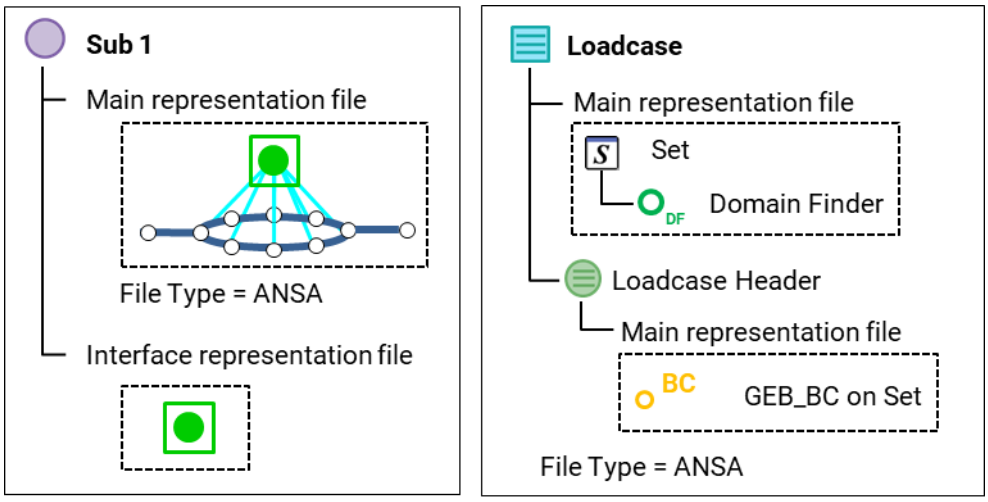




On Save in DM or Output, the Transfer Function Subcase is expanded and creates a separate Subcase per each active DOF of the LC_POINTs that are collected by the Domain Finder.

```
$ BEGIN CASE CONTROL
$
ECHO = SORT
$
SUBCASE 1
LABEL = FRF_LOAD_DOF 10000019:1:SEAT_FRONT_L2
DLOAD = 1
$
SUBCASE 2
LABEL = FRF_LOAD_DOF 10000019:2:SEAT_FRONT_L2
DLOAD = 2
$
SUBCASE 3
LABEL = FRF_LOAD_DOF 10000019:3:SEAT_FRONT_L2
DLOAD = 3
$
SUBCASE 4
LABEL = FRF_LOAD_DOF 10000020:1:SEAT_FRONT_L1
DLOAD = 4
$
SUBCASE 5
LABEL = FRF_LOAD_DOF 10000020:2:SEAT_FRONT_L1
DLOAD = 5
$
SUBCASE 6
LABEL = FRF_LOAD_DOF 10000020:3:SEAT_FRONT_L1
DLOAD = 6
$
SUBCASE 7
LABEL = FRF_LOAD_DOF 10000023:1:SEAT_FRONT_L3
DLOAD = 7
$
SUBCASE 8
LABEL = FRF_LOAD_DOF 10000023:2:SEAT_FRONT_L3
DLOAD = 8
$
SUBCASE 9
LABEL = FRF_LOAD_DOF 10000023:3:SEAT_FRONT_L3
DLOAD = 9
$ END CASE CONTROL
```

The Module and Loadcase contents stored in DM, are described in the image below:



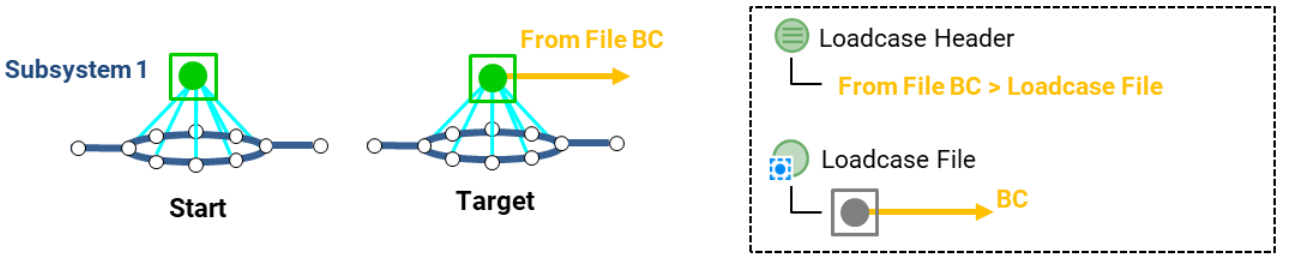
Therefore, a Loadcase Header with a Transfer Function Subcase can be applied to Simulation Models that contain different Loadcase Points, given that the Domain Finder can be re-applied to fill the corresponding Set.

7.2.5. File Loadcase

Through the Loadcase Header it is possible to utilize boundary conditions (LOADs, LOAD SETs, CONSTRAIN), frequency ranges (FREQUENCY), modes (METHOD) and time step (TIME STEP) definitions that are available in a Nastran deck file.

In the Modular Run Environment:

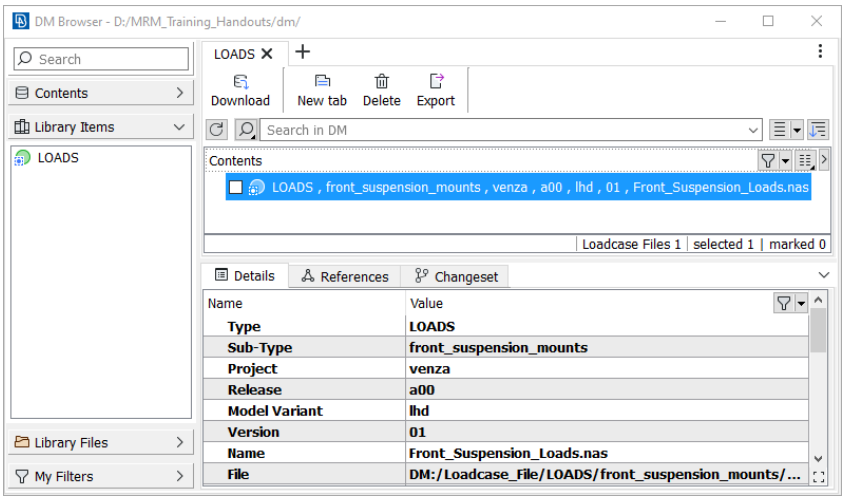
- The Nastran deck file is stored in DM as a library item (Loadcase File or Library File)
- The Loadcase Header contains references to the Nastran deck file though FILE LOADCASE entities
- The FILE LOADCASE can filter entities of the Nastran deck file based on type and name.
- The entities filtered by a FILE LOADCASE can be utilized by the Loadcase Header upon Build
- The Nastran deck file may contain undefined Loadcase Points, to take advantage of the automatic attachment of Point Boundary conditions as described in 7.2.1.7.



Typical From File BC implementation in ANSA

7.2.5.1. Library item in DM

The Nastran deck file shall be added in DM as a library item. Either the Loadcase File or the Library File library item type can be used.

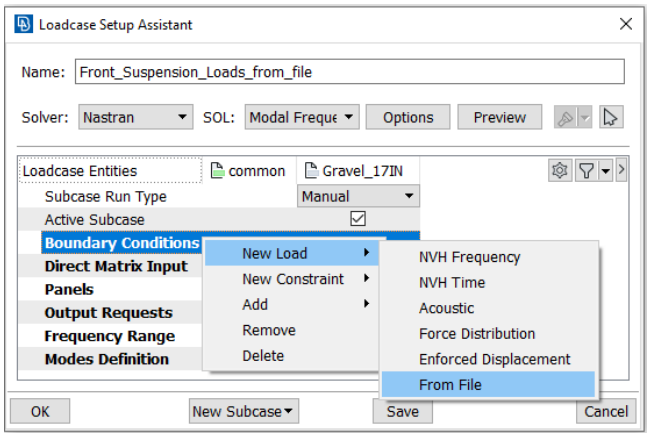


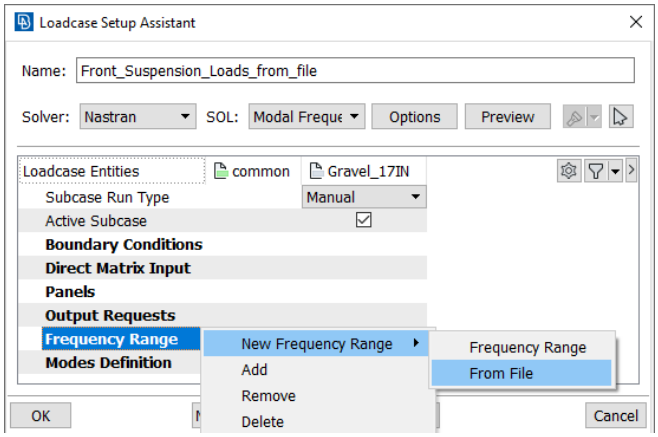
The contents of the library item that can be utilized by the Loadcase Header are the entity types supported by the FILE LOADCASE entity (please refer to paragraph 7.2.5.2).

```
$ FREQ for  DLOAD: 100 Gravel_17IN
FREQ1,100,1.,1.,99
$ DLOAD,100,Gravel_17IN
DLOAD,100,1.,1.,10001,1.,10002,1.,10003,+
+,1.,10004,1.,10005,1.,10006,1.,10011,+
+,1.,10012,1.,10013,1.,10014,1.,10015,+
+,1.,10016,1.,10021,1.,10022,1.,10023,+
+,1.,10024,1.,10025,1.,10026,1.,10031,+
+,1.,10032,1.,10033,1.,10034,1.,10035,+
+,1.,10036,
RLOAD1,10001,10001,0.,0.,1000111,1000112,0
DAREA,10001,2000001,1,1.
TABLED1*1000111         LINEAR          LINEAR                          *
*                                                                       *
*       0.00000000E+00  0.00000000E+00  2.00000000E+00  0.00000000E+00  *
*       4.00000000E+00  0.00000000E+00  6.00000000E+00  0.00000000E+00  *
*       ENDT
```

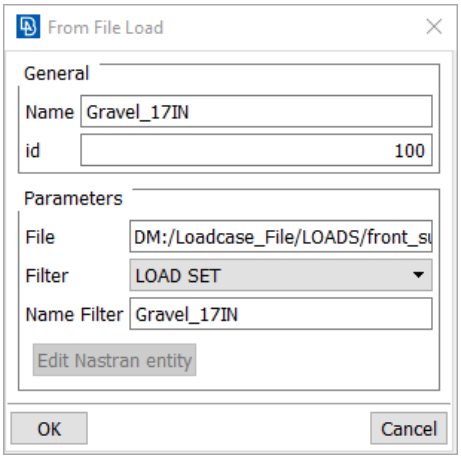
7.2.5.2. Creating 'From File' Loadcase entities

Different types of loadcase entities can be created using the 'From File' option in the Loadcase Header:





A FILE LOADCASE entity can be configured in the pop-up window, by specifying the fields below:



Name: The name of the loadcase entity in the Loadcase Header. It is also reflected to the corresponding FILE LOADCASE.

Id: The Id of the FILE LOADCASE

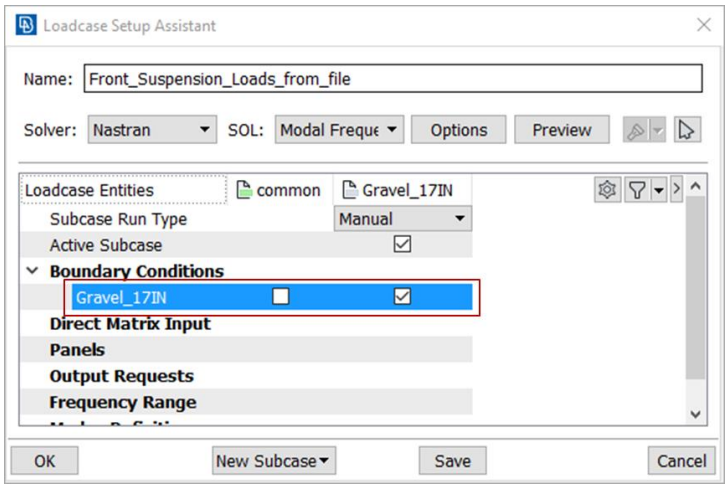
File: The path to the library item in DM

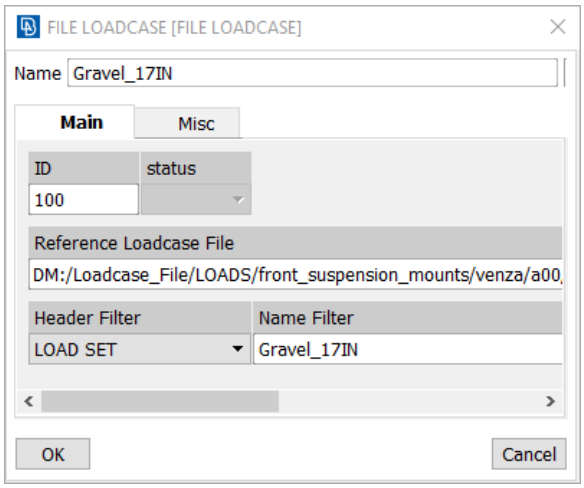
Filter: Control the type of entities contained in the library item that will be utilized by FILE LOADCASE. Each value in the **Filter** drop down menu corresponds to different Nastran keywords:

Filter	Nastran Keyword
LOAD	RLOAD, TLOAD, DAREA, LOAD SET, FORCE, ACCEL1, PLOAD4, SPCD, RFORCE, GRAV
LOAD SET	DLOAD, LOAD
CONSTRAIN	SPCSET, SUPORTSET, SPC, SPC1, SUPORT
FREQUENCY	FREQSET
TIME STEP	TSTEP
METHOD	METHOD

Name Filter: Optionally filter the entities of the library item that are utilized by FILE LOADCASE by name. An exact match with the provided string is initially searched for. In case that exact match is not detected, then all the entities whose name contains the provided string are filtered.

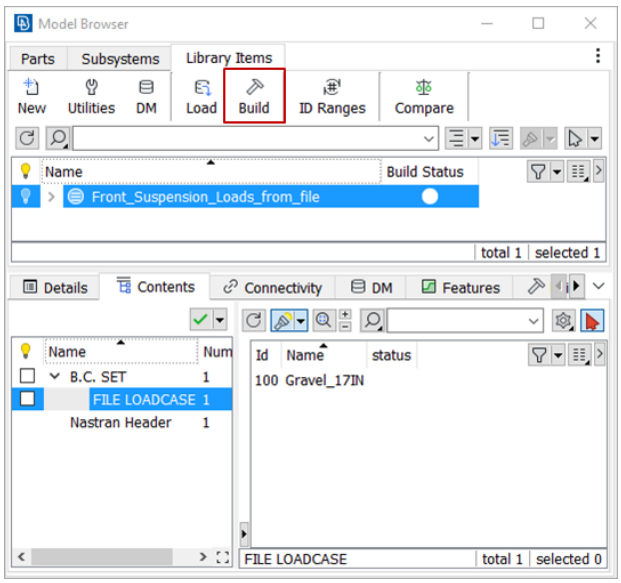
Upon confirmation, a 'From File' loadcase entity is added in the Loadcase Header. This loadcase entity can be activated in one or more Subcases. The corresponding FILE LOADCASE is created in the Database Browser (B.C SET). Initially the status of the FILE LOADCASE is blank, to note that the FILE LOADCASE is not applied.

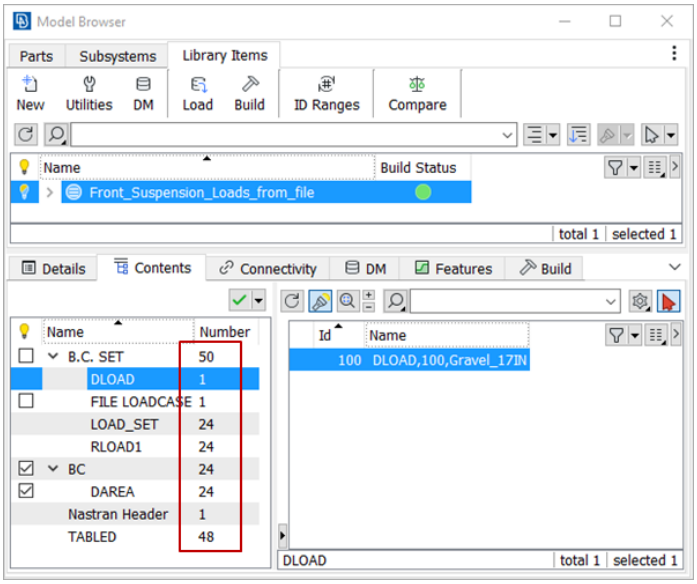




7.2.5.3. Applying a FILE LOADCASE

The FILE LOADCASE is automatically applied during the **Build** of the Loadcase Header or can be manually applied through the respective context menu option at any time. Upon application, all the filtered entities of the selected library item are loaded in ANSA and are added to the Loadcase Header.





Note that any entities used by the filtered entities are also loaded. For example, in case that the filtered entity is a DLOAD, any related RLOAD/DAREA/TABLED etc entitites are loaded too.

If un-defined Loadcase Points are present in the library item, these are merged with defined Loadcase Points present in the model as described in 7.2.1.7. (composition during Load). In this way it is possible to apply boundary conditions on specific points.

A FILE LOADCASE can be erased through the respective context menu option, in the Database Browser. Upon **Erase** all the previously loaded entities are removed.

7.3. Creating a Loadcase from Target Points

Pedestrian protection and occupant interior safety simulations require the application of the same loading scenario on several different positions of the model. In order for an analyst to study the performance of a particular Simulation Model version for pedestrian and occupant safety, several Simulation Runs must be studied, having the test device (headform or legform) hit a different target location each time. All the different Simulation Runs differ only in the following characteristics:

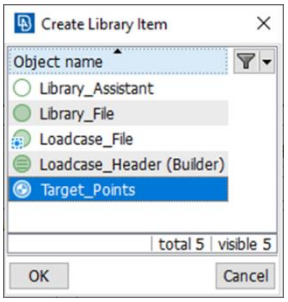
- Position of the test device: The test device is positioned at a different location and, in some cases, at a different impact angle, based on the Target Point being selected
- Velocity of the test device: For some particular loadcases, a different initial velocity is determined for each target point

ANSA offers specific tools for the pre-processing of such Loadcases that are available under SAFETY>Pedestrian and SAFETY>Interior. More or less, the process followed by these tools consists of two main steps:

- Create the Target Points
  Depending on the protocol and on user-defined settings, several Target Points are generated
- Position the test device and output the main file
  Depending on the protocol and on user-defined settings, the test device is positioned on all or on a subset of the Target Points extracted in the previous step. For each point, a solver main file is output, containing the appropriate transformation card for the positioning of the test device and the right initial velocity, if required.

Mapping this process in the Modular Run Environment requires the organization of all related data in appropriate Model Browser Containers and the association of the process steps described above with the Build Process of the Loadcase. The paragraph below describes how Target Points-based loadcases are managed in the Modular Environment.

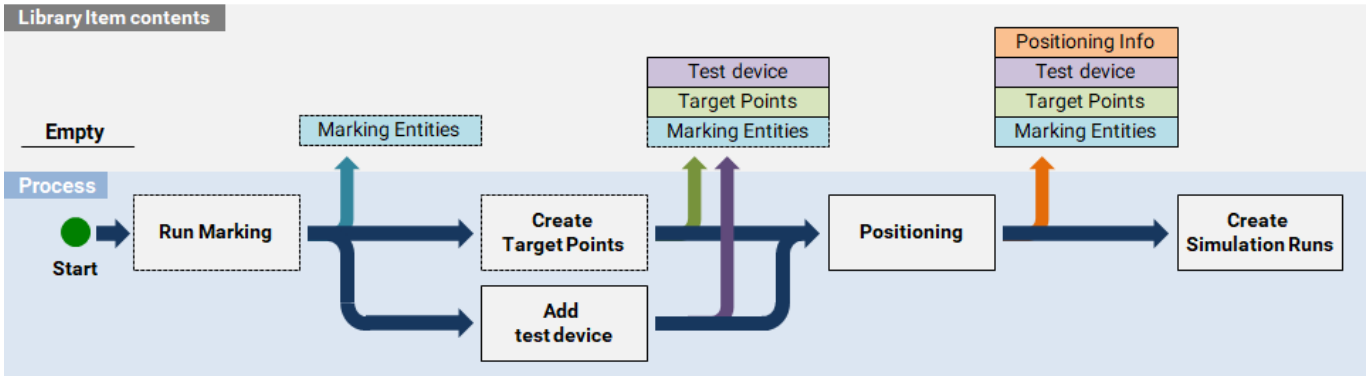
7.3.1. Target Points Library Item



The Modular Run Environment offers a specific type of Library Item that handles loadcases based on Target Points. This Library Item can be created in the Library Items tab in the Model Browser through the option New > From scratch > Target_Points.

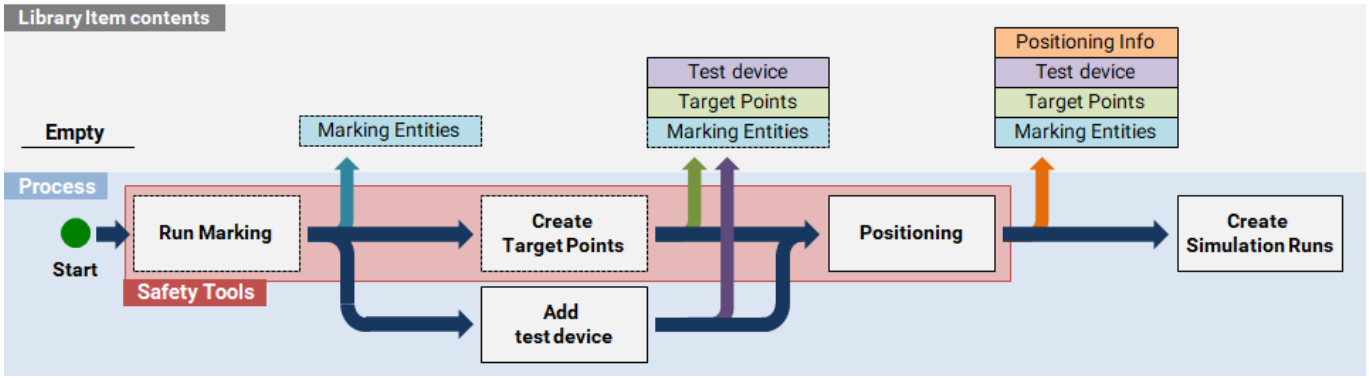
The key characteristic of this Library Item is that it is associated with a group of Target Points and it can use them as source for generation of Simulation Runs. One Simulation Run for each Target Point.

This Library Item goes through several stages in its life-cycle, as shown in the diagram below. The light blue lane at the bottom represents the different steps of the process. The light grey lane at the top represents the data assigned to the Library Item.



- In the beginning, the Library Item is empty.
- After running the marking process, the marking entities are generated (curves, elements, 3D-points) and are all assigned to the Library Item.
- In the next step, the Target Points are generated and are assigned to the Library Item.
- Additionally, the test device is associated with the Library Item.
- Next, the positioning process is executed and the transformation information that is required to place the test device at each target is stored under the Library Item
- In the end, the Library Item has all the required information in order to produce Simulation Runs.

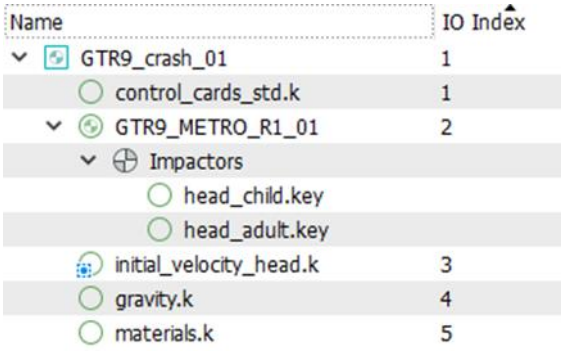
The pedestrian protection tool is integrated in the Modular Run Environment in order to carry out certain steps of the process, as shown in the diagram below (The occupant safety Interior 201U tool will also be integrated soon)



Note that the *Run Marking* and *Create Target Points* steps are outlined with a dashed line because they could be skipped in case the Target Points are read from some external source.

The key characteristic of pedestrian protection pre-processing in the Modular Environment is the possibility to store all positioning information in the Target Points Library File in DM and then use it as required in order to produce Simulation Runs for selected Target Points “just in time”.

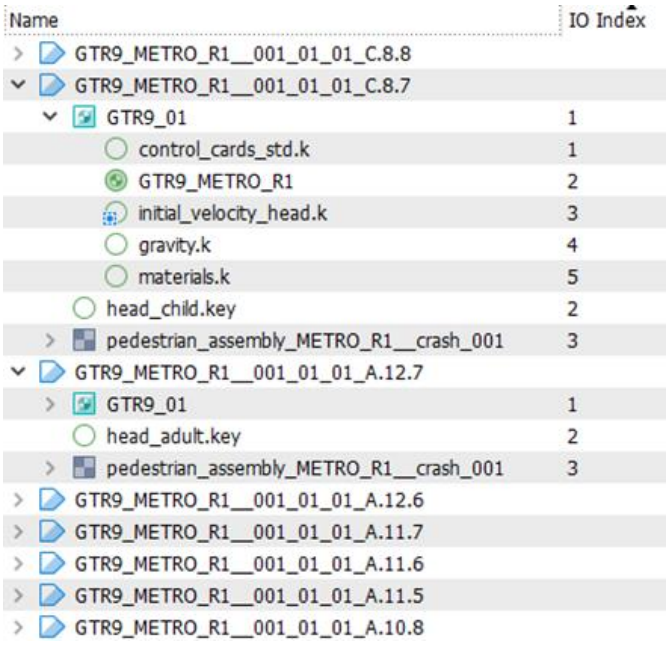
The screenshot below shows a Pedestrian Loadcase set-up for protocol GTR9/UN-R 127. The peculiarity of this protocol is that the assessment is done on a raster consisting of Target Points for adult and child headform.



The Target Points Library Item is the one named GTR9_METRO_R1_01. Within the Loadcase this item is combined with standard Library Items that represent the control cards, the initial velocity, the gravity and the materials. The Impactors (test devices) are associated with the Target Points Library Item. Due to the peculiarity of this loadcase, two impactors are added.

The *DM/IO Index* attribute of the adapters has been used in order to control the order in which the include statements for these Library Items will appear within the Loadcase.

After building the Loadcase and selecting the targets for which a Simulation Run should be created, the different Simulation Runs are created.

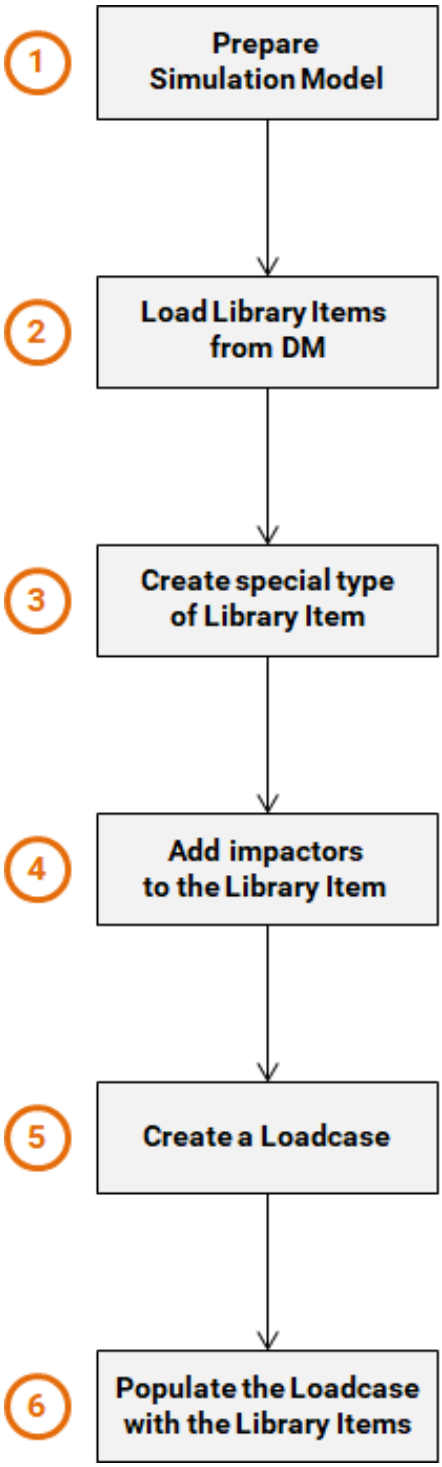


In the screenshot on the left, 8 Simulation Runs have been created, 5 on targets for child heads and 3 on targets for adult heads (can tell by the name of the Target Point that is included as a last attribute for the composition of the run names. Adult targets start with an “A” whereas child targets start with “C”)

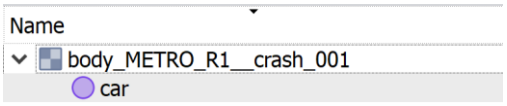
Each Run consists of the Loadcase, the Simulation Model and one direct reference to the test device in use, either the header_adult.key or the header_child.key. Notice that the Target Points Library Item appears as having no contents. Its contents, or more precisely the right content each time, are passed to the Simulation Run.

7.3.2. Step-by-step set-up of a Pedestrian Loadcase

The set-up of a pedestrian Loadcase is a 10-step process.



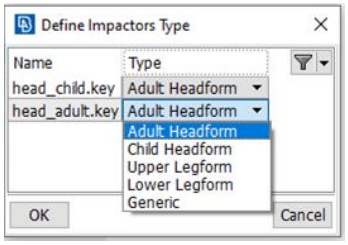
Step 1: The analyst must prepare the Simulation Model. The Simulation Model may consist of one or more Subsystems and Library Items as shown below.

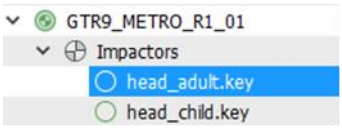


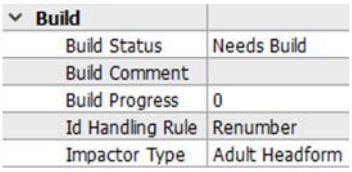
Step 2: All files that will be needed for the loadcase, e.g. control cards, contact cards, materials, etc. must be added as Library Items in the respective tab of the Model Browser. The option *New>From DM* can be used to load items stored already in the library.

Step 3: A new Target Points Library Item must be created through the option *New>From scratch* selecting the item *Target Points* from the list.

Step 4: One or more impactors must be added to this Library Item through the context menu option *Actions> Impactors from DM*. After marking the Library Items in the DM Browser and pressing **Download**, the dialog on the left pops-up, in order for the analyst to define the Impactor Type. The Impactor Type is finally stored as a Build Attribute of the Library Item.



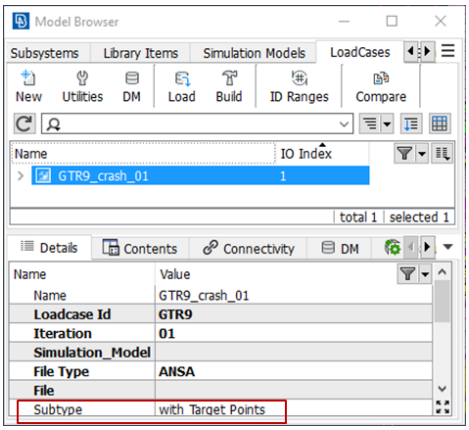




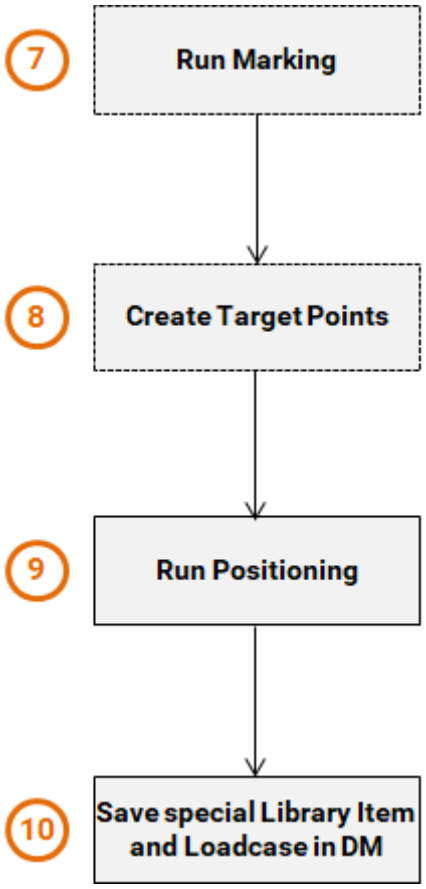
At this point, if the analyst needs to add any adaptation for the ids of the impactors it is possible through the context menu option *Actions>Target Id Ranges*.

Step 5: Create a new Loadcase in the Loadcases tab

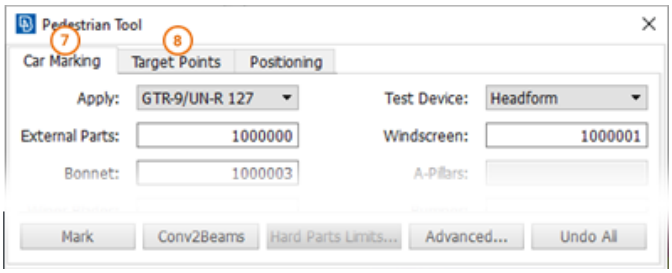
Step 6: Populate the new loadcase with the Library Items created in steps 2 and 3 with drag and drop or through the table view. Mind to define the proper DM/IO Index values to control the order of include statements in the main file.



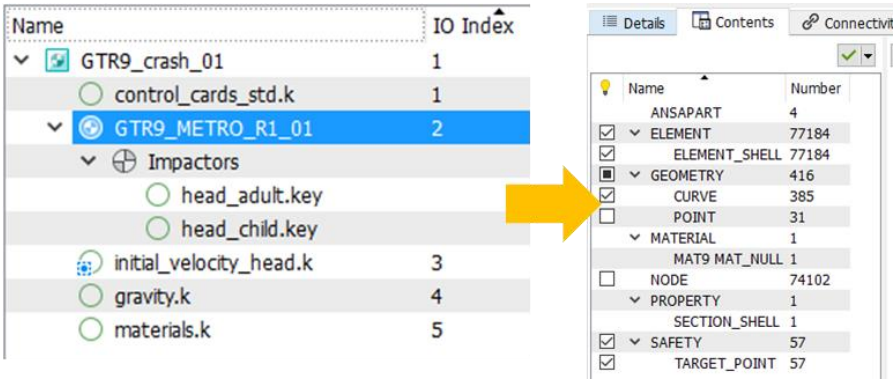
Notice that as the new Loadcase is populated with a Target Points Library Item, its Subtype changes to *with Target Points*.



Steps 7 and **8** are carried out by the Pedestrian tool. Launch the pedestrian tool through the option *Actions>Safety Tool* on the Target Points Library Item.

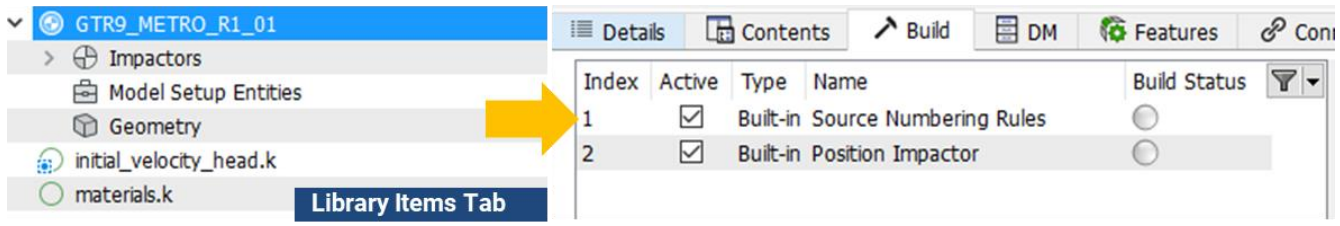


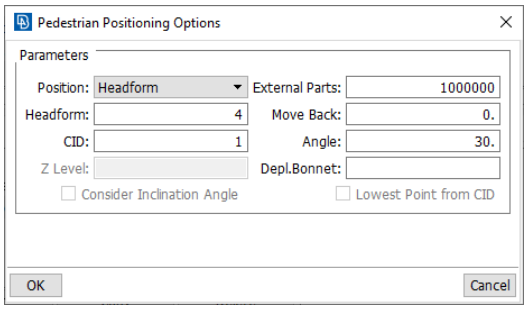
Any entities created during the Car Marking and Target Points creation process are added to the contents of the Library Item.



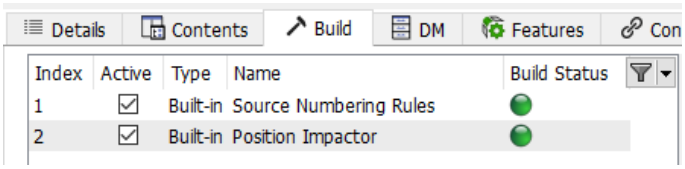
In case Target Points are imported from some external source, they just need to be dragged and dropped on the Target Points Library Item.

From this point on, the Build process of the Library Item can be used.





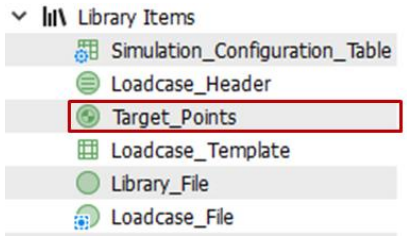
Step 9: The Build Action **Position Impactor** will load the impactor and open the *Pedestrian Positioning Options* window, where the user needs to set the inputs for the positioning process. Pressing OK in this window, positioning is performed for all Target Points.



Step 10: At this point, the Target Points Library Item can be saved in DM.

Its default primary attributes are:

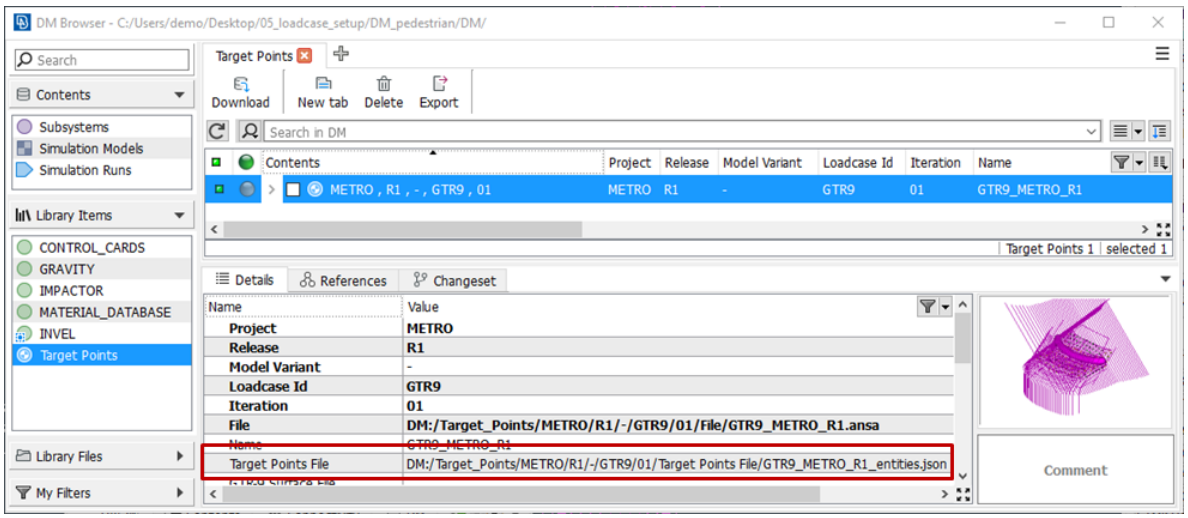
- **Loadcase Id**: Should describe the type of loadcase that this Library Item will be used for (e.g. head-GTR9, head-euncap, upper leg, lower leg, etc.)
- **Project**: The Project of the model
- **Release**: The Release of the model. The same Target Points Library Item may be usable for more than one Releases, if there's no change in styling characteristics
- **Model Variant**: The Variant of the model. The same Target Points Library Item may be usable for more than one Variants, if there's no change in styling characteristics
- **Iteration**: The version of the Library Item. The analyst may need to add more Target Points manually. These will be stored as a new Iteration.



The names of the primary attributes as well as the setup of any closed lists of predefined values may be configured through the DM Schema Editor under the DM Object Type **Target_Points**

By default, this item is only saved in DM as an ANSA file.

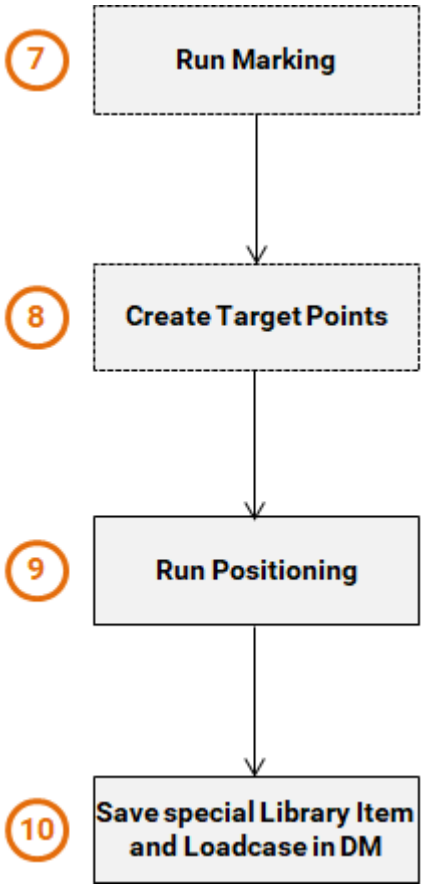
At the moment of save, an additional file that contains information on the target points is saved in json format and uploaded to DM under the Additional Attribute **Target Points File**.



Additionally, at this point the Loadcase must also be saved in DM through the Loadcases tab. Loadcases of this type are automatically saved in DM as ANSA hierarchy files.

7.3.3. Step-by-step set-up for a FMVSS201U Loadcase

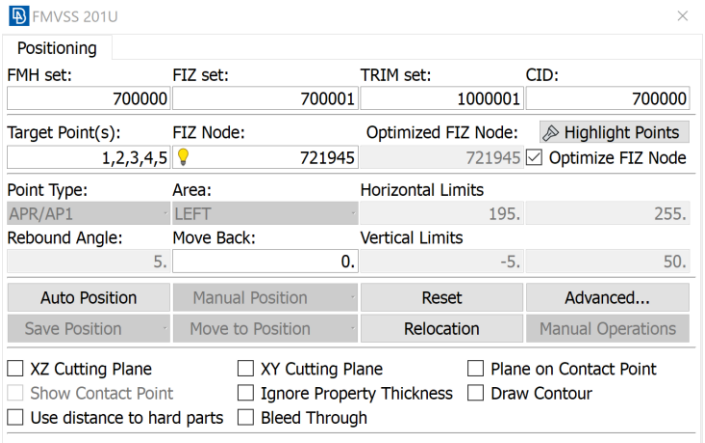
The set-up of a FMVSS201U Loadcase is a process nearly identical with the Pedestrian Loadcase process described above. From step 1 to step 6, where the Loadcase is populated with the Library Items and its Subtype changes to *with Target Points*, repeat the process as described on 7.3.2.



Steps 7 and **8** are carried out by the Interactive FMVSS201U tool. Launch the FMVSS201U Interactive Marking through the option *Actions>Safety Tool* on the Target Points Library Item. The user needs to define the parameters by pressing the respective Parameters button and then press Run or Run Chain (creates the dependencies of the selected block too) to create the Target Points.

Any entities created during the Car Marking and Target Points creation process are added to the contents of the Library Item. From this point on, the Build process of the Library Item can be used.

Step 9: The Build Action **Position Impactor** will load the impactor and open the *Positioning* window, where the user needs to set the inputs for the positioning process. Pressing OK in this window, positioning is performed for all Target Points.

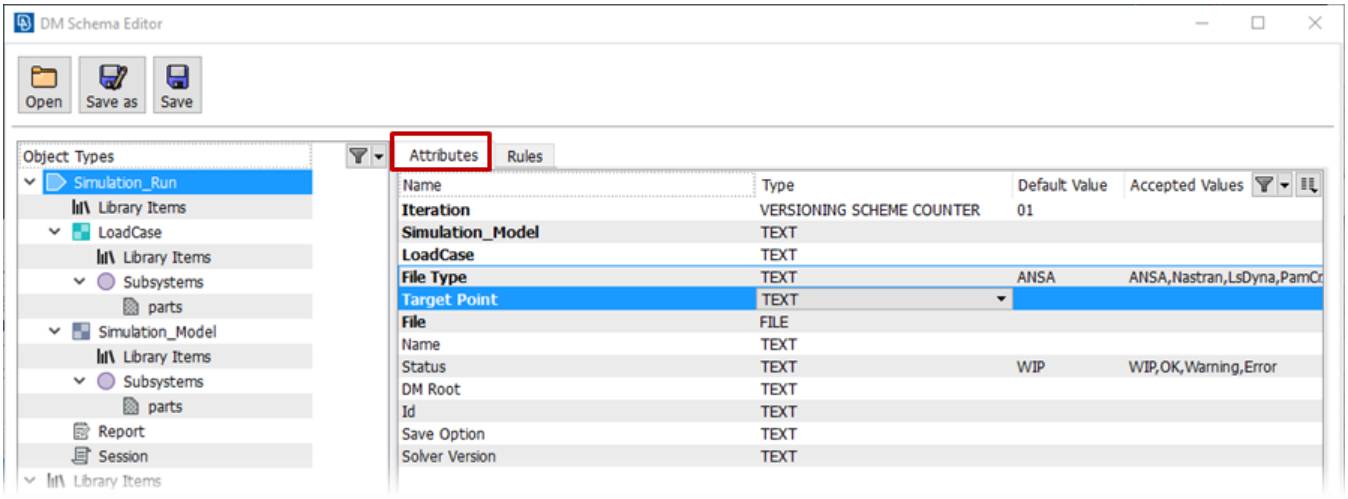


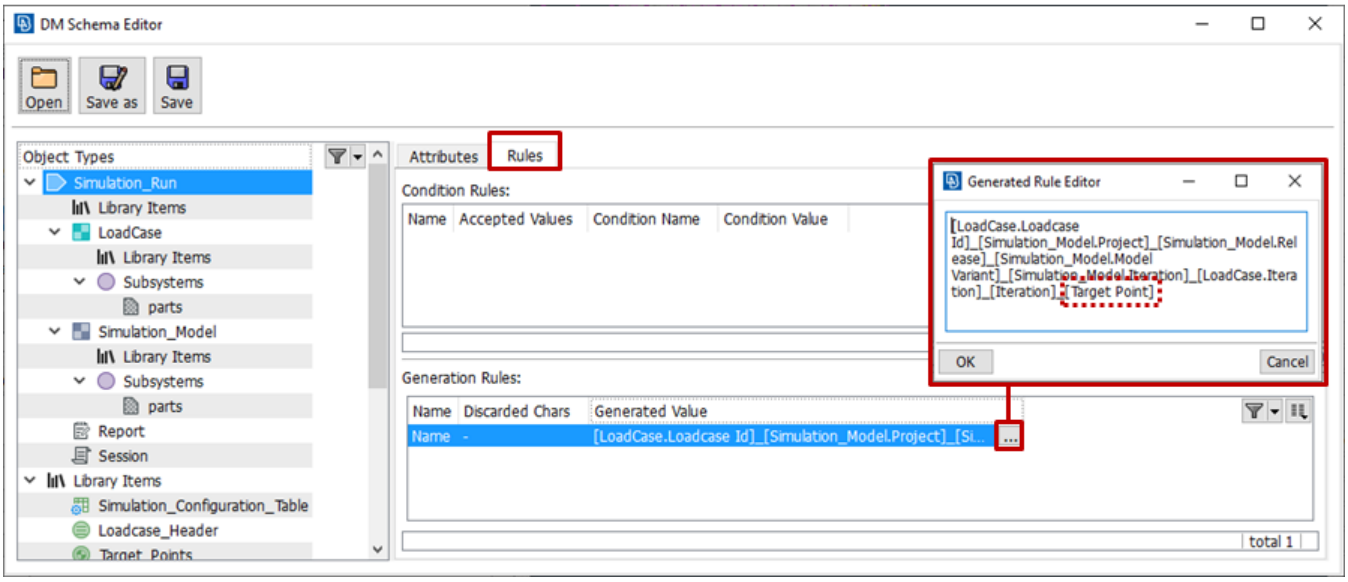
Step 10: At this point, the Target Points Library Item can be saved in DM. By default, this item is only saved in DM as an ANSA file.

An additional file that contains information on the target points is saved in json format and uploaded to DM under the Additional Attribute **Target Points File**.

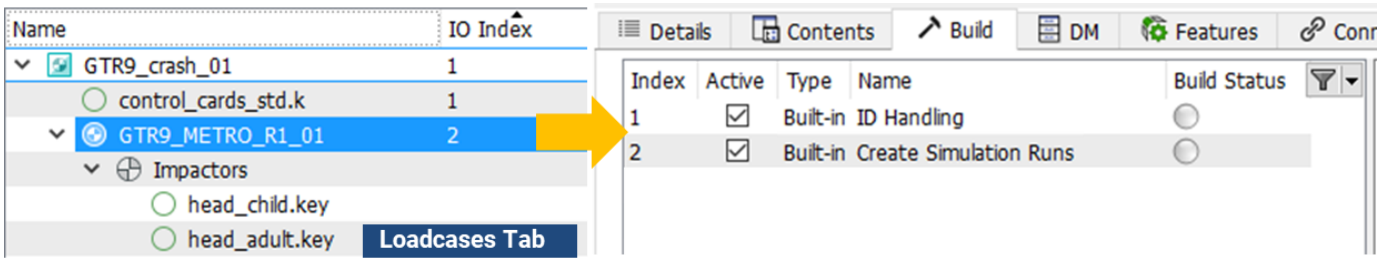
7.3.4. Creating Simulation Runs from a Target Points Loadcase

Before the creation of the Simulation Runs, the analyst must add a primary attribute with name **Target Point** in the Simulation Run definition in the **DM Schema Editor** and also add it to the rule used for the composition of the Simulation Run Name. This attribute is mandatory for the identification of the different runs that will be generated down the line. A Simulation Run properly configured for use with Target Points Loadcases is shown below:

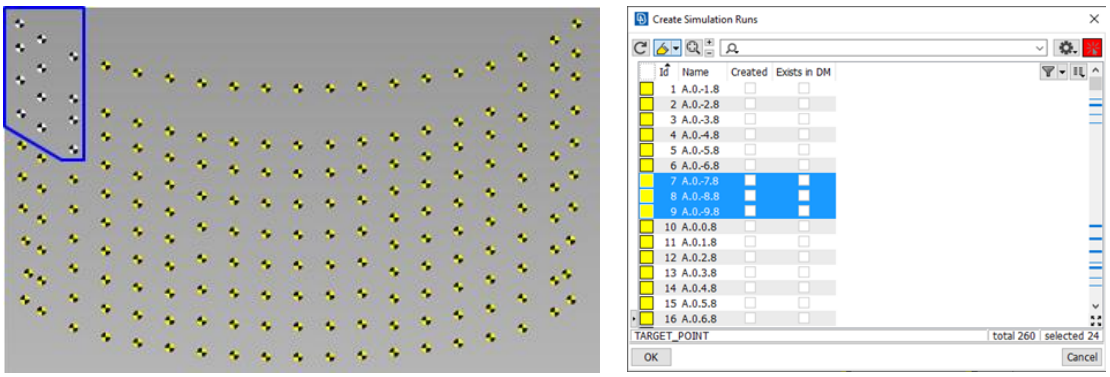




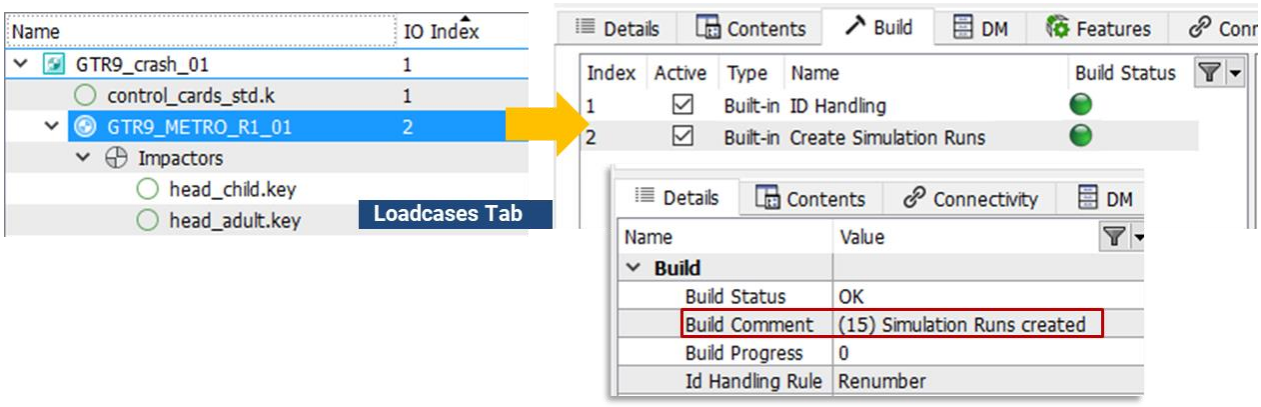
Building the Loadcase will execute the adapter Build Action **Create Simulation Runs**.

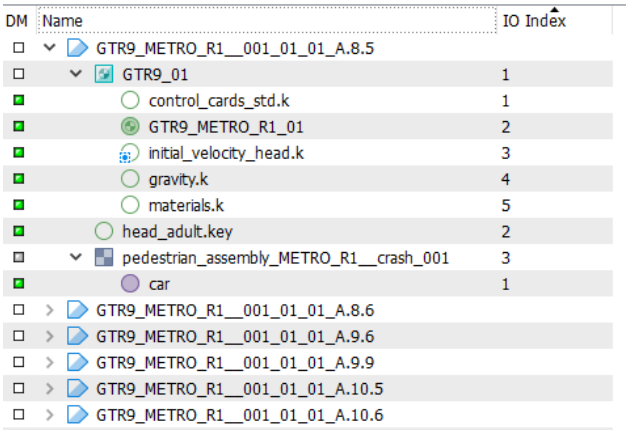


This action will open the *Create Simulation Runs* window where the analyst can select the Target Points for which a new Simulation Run must be created. Selection can be done from the list or graphically.



Pressing OK, one Simulation Run will be created *for each selected* Target Point and a Build comment will be added to the Loadcase Adapter, with information about the number of Simulation Runs created.

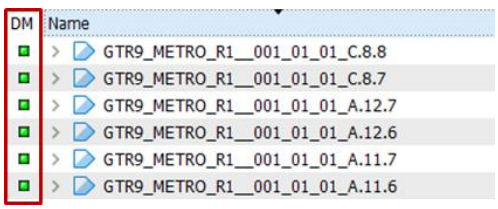




The Build Process would get OK status even in case Cancel is pressed in the *Create Simulation Runs* window. However, in that case, the comment would be *(0) Simulation Runs created.*

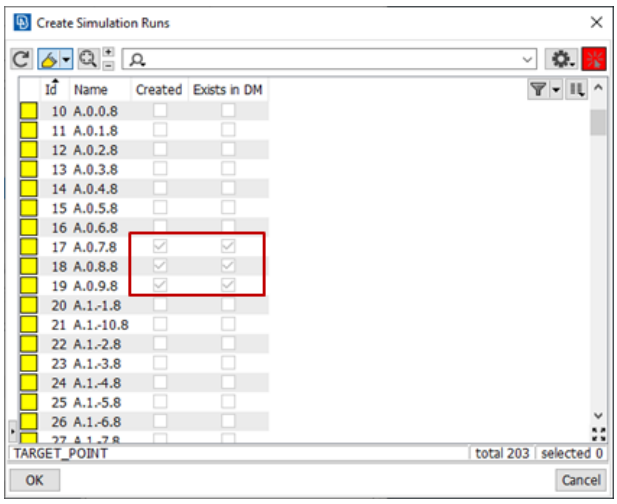
This is the list of the Simulation Runs created. Each run contains a reference to the Loadcase and the Simulation Model, and the proper impactor each time. The impactor is selected according to the requirements of each protocol and according to the Impactor Type specified during the Loadcase creation process.

Running **Save in DM** for the multi-selection of runs will push them to DM.



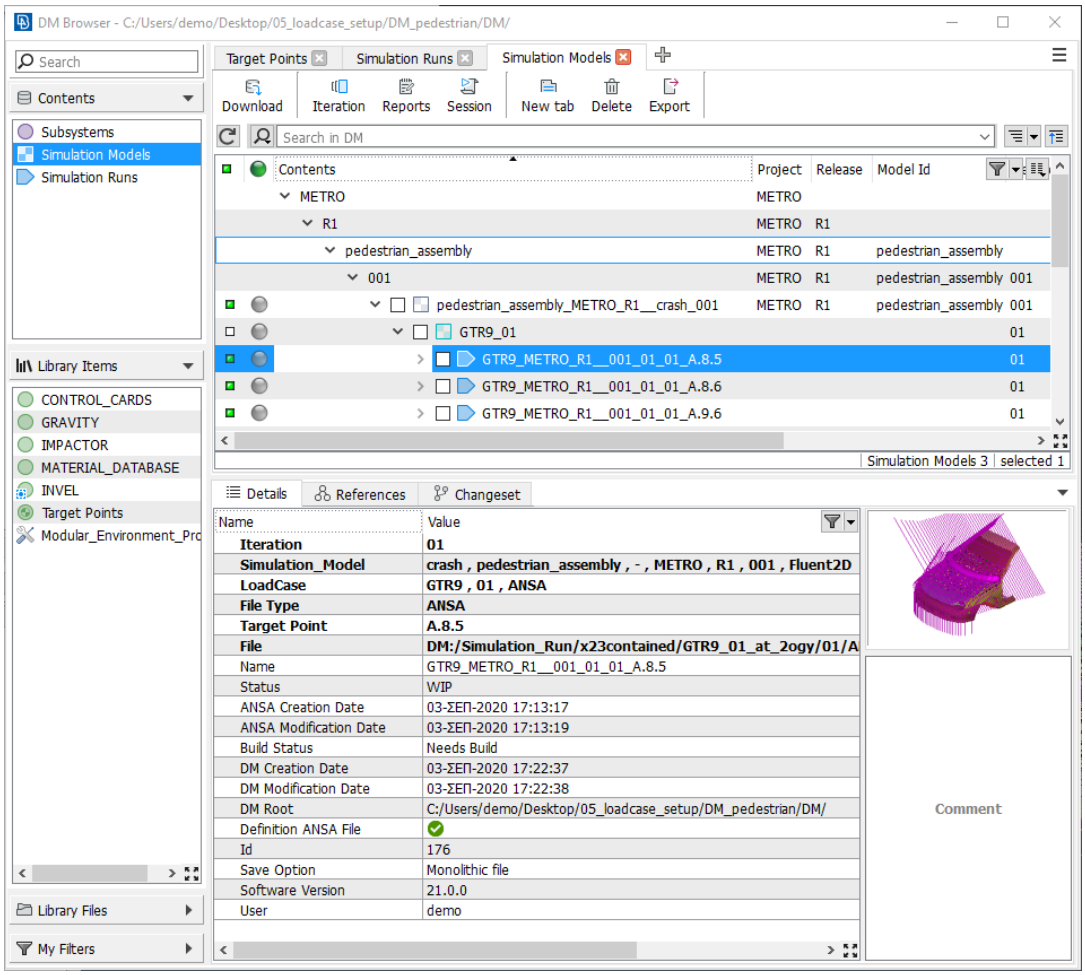
The **DM Update Status** becomes green, indicating that the runs have been just saved in DM.

At this point, in order to add some more Simulation Runs, the user would have to **Build** the Build Action **Create Simulation Runs.**



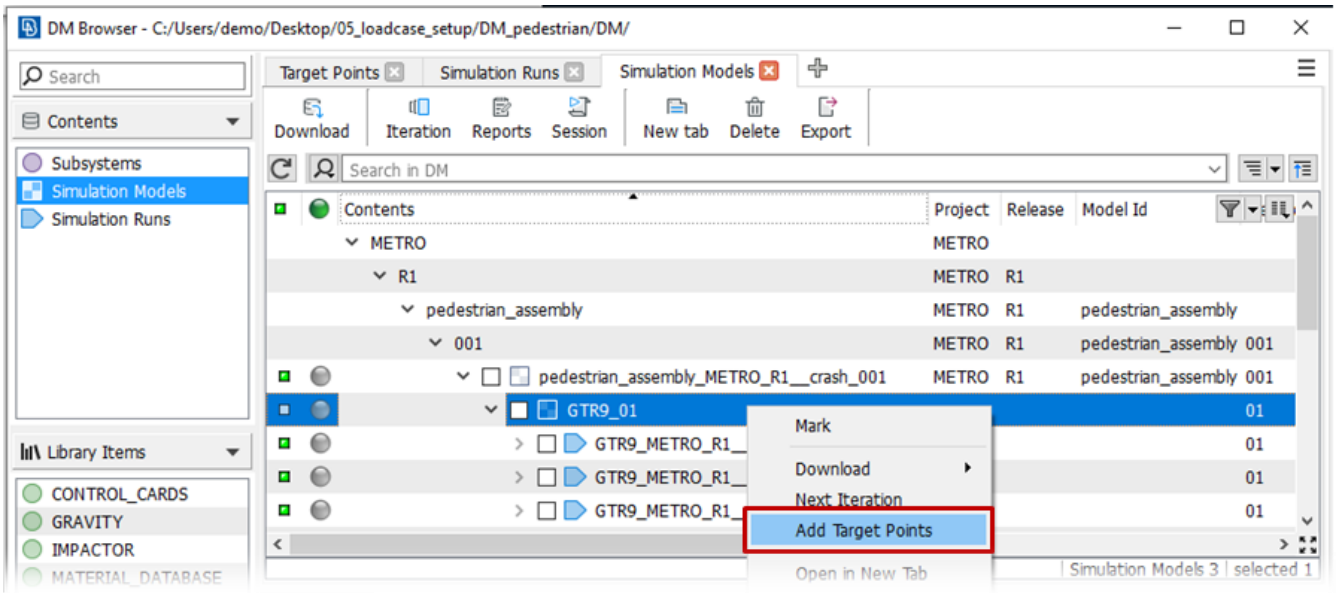
The Create Simulation Runs window marks the Target Points that are already represented by a Simulation Run in the ANSA session and in the data repository in the columns *Created* and *Exists in DM* respectively.

In the *DM Browser*, the Simulation Runs are listed in the Simulation Runs tab. However, a more convenient view is the one offered through the Simulation Models tab.



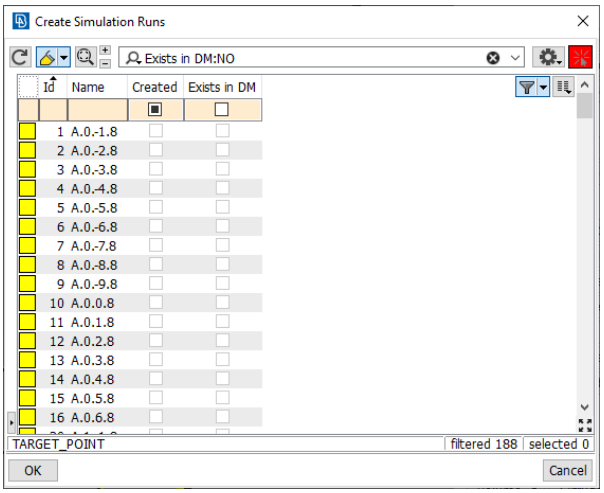
7.3.5. Adding more runs under an existing Target Points Loadcase

In order for an analyst to produce new Simulation Runs for some Target Points that have not been simulated yet, it's possible to use the **Add Target Points** context menu option on the Loadcase DM Object in the *DM Browser*.

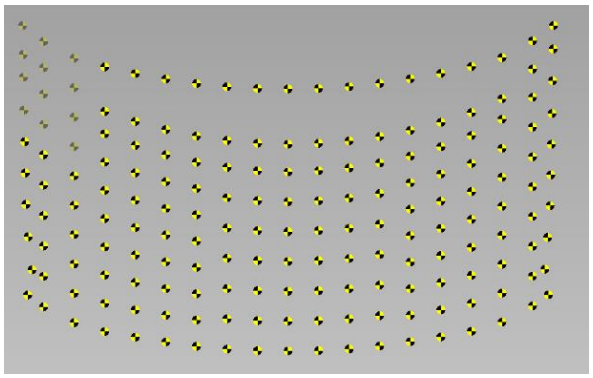


The definition of the Loadcase and the Simulation Model are loaded in the ANSA session.

At this point, the analyst should **Build** the Loadcase in order to get the *Create Simulation Runs* window and select the Target Points for which new Simulation Runs will be created. Note that a filter can be added in order to quickly identify the targets that are not yet saved in DM as Simulation Runs.



Notice that the 12 Target Points at the top left of the picture are not eligible for selection, since they already exist in DM.



7.3.6. Structure of the Simulation Run main file

The structure of the Simulation Run main file is presented in the table below, taking an LS-DYNA run as an example. It's split in blocks, for clarity.

Block 1: Simulation Run DM header

```
$============================DM INFO BEGIN============================
$ANSA_DM_INFO_PRE_FORMAT;
$
$Entity Type      : Simulation Run
$Iteration        : 01
$Simulation_Model : 149
$LoadCase         : 153
$File Type        : LsDyna
$Target Point     : C.7.9.8
$File             :
$Name             : GTR9_METRO_R1__001_01_01_C.7.9.8
$User             : demo
$Solver Version   : R11.1
$
$END_ANSA_DM_INFO_PRE_FORMAT;
$============================DM INFO END============================
*KEYWORD
```

Block 2.1: Loadcase DM header

```
$============================DM INFO BEGIN============================
$ANSA_DM_INFO_PRE_FORMAT;
$
$Entity Type      : Loadcase (LoadCase)
$Loadcase Id      : GTR9
$Iteration        : 01
$Simulation_Model :
$File Type        : ANSA
$File             : GTR9_crash_01.ansa
$Name             : GTR9_crash_01
$User             : demo
$Special Type     : with Target Points
$Solver Version   : R11.1
$
$END_ANSA_DM_INFO_PRE_FORMAT;
$============================DM INFO END============================
```

Block 2.2: Loadcase content

```
*INCLUDE
//data/pedpro/DM/Library_File/CONTROL_CARDS/-/01/control_cards_std.k/File +
/control_cards_std.k
*INCLUDE
//data/pedpro/DM/Loadcase_File/INVEL/pedestrian_head/-/-/-/01/initial_velo +
city_head.k/File/initial_velocity_head.k
*INCLUDE
//data/pedpro/DM/Library_File/GRAVITY/-/01/gravity.k/File/gravity.k
*INCLUDE
//data/pedpro/DM/Library_File/MATERIAL_DATABASE/-/01/materials.k/File/mat + erials.k
$ANSA_INCLUDE;END;
*INCLUDE
//data/pedpro/DM/Simulation_Run/x23contained/GTR9_crash_02_at_psxy/01/LsD +
yna/C.7.9.8/External_Single_Transformation_File_Abs_Paths/auto_created_tr +
ansf_abs_head_child.key
```

Block 3.1: Simulation Model DM header

```
$============================DM INFO BEGIN============================
$ANSA_DM_INFO_PRE_FORMAT;
$
$Entity Type    : Simulation Model
$Discipline     : crash
$Model Id       : pedestrian_assembly
$Model Variant  : -
$Project        : METRO
$Release        : R1
$Iteration      : 001
$File Type      : LsDyna
$File           :
$Name           : pedestrian_assembly_METRO_R1__crash_001
$User           : demo
$Solver Version : R11.1
$
$END_ANSA_DM_INFO_PRE_FORMAT;
$============================DM INFO END============================
$
```

Block 3.2: Simulation Model content

```
*INCLUDE
//data/pedpro/DM/Subsystems/car/-/-/-/001/-/LsDyna/common/repr/car.key

*END
```

In block 2.2 the last include statement references the headform through an adaptation metafile that was created to hold the transformation. In this file, the source Library Item of the headform is transformed with the transformation keyword created by ANSA.The content of this file is shown below:

auto_created_transf_abs_head_child.key

```
*KEYWORD
$ANSA_COMMENT;1;DEFINE_TRANSFORMATION; Pedestrian Include File Parameters~
$ Target Point              : C.7.9.8~
$ Target Point ID           : 203~
$ Impact Angle              : 50.00~
$ Target Point Coordinates  : -3499.322, 666.975, 870.512~
$ Contact Point Coordinates : -3546.020, 663.768, 869.170~
$ Contact Distance          : 46.827~
$ Impactor DX, DY, DZ       : -7442.293, 969.712, 73.769~
$~
$~
$~
$;
*DEFINE_TRANSFORMATION_TITLE
Anonymous *DEFINE_TRANSFORMATION
         1
ROTATE            0.        1.        0.   3891.31  -302.737 858.3097550.0010885
TRANSL    -7442.2925969.711951 73.768952
*INCLUDE_TRANSFORM
//data/pedpro/DM/Library_File/IMPACTOR/head_child/01/head_child.key/File/ +
head_child.key
         0         0         0         0         0         0         0
         0
                                                 0
         1
*END
```

7.4. Interface information for post-processing

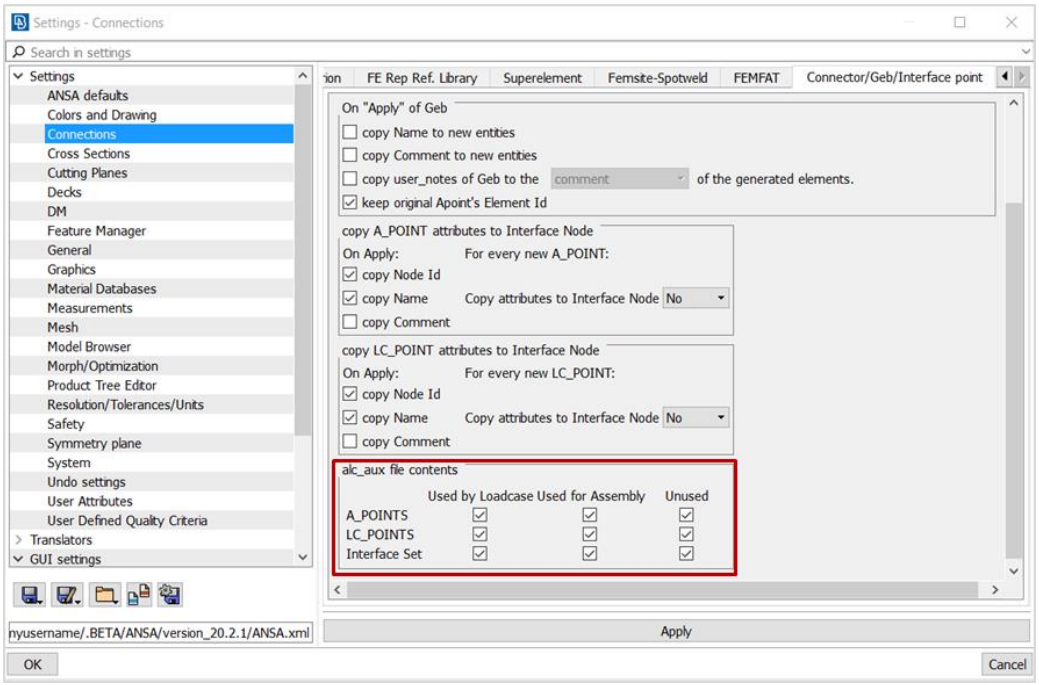
As described in Chapters 6 and 7, Interface Entities (A_POINTs, LC_POINTs and Interface Sets) are used to mark locations for Assembly and Loadcase setup. The same approach can be used for marking locations for post-processing in META, as it enables the effortless, id-independent, identification of model entities of interest. Interface Points in META can be used for:

- The visualization of Interface Nodes on the 3D model
- The identification of nodes on 2D and 3D
- The creation of a display model

Interface Entities created in ANSA can be passed to META through the ANSA comments. However, as, in most of the cases CAE analysts read in META the binary result files (e.g. op2, odb, d3plot, etc.) and not the solver keyword files that can include ANSA comments, a supplementary file, with the extension *.alc_aux* is exported along with the main representation file during the output of models that contain interface information. Using this file, which is written in "ANSA-comments" format, there is no need to keep track of the interface information in other ways such as the entity Name, Id etc.

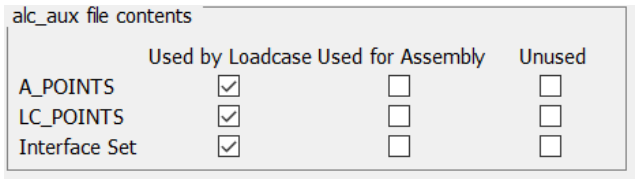
7.4.1. Controlling the contents of the alc_aux file in ANSA

It is possible to control the contents of the *alc_aux* file of Simulation Runs based on the type of the Interface Entities and the context, i.e. whether they are used for Assembly or for Loadcase set-up. These controls are available in the Settings window, under *Connections > Connector/Geb/Interface point* tab.



Used by Loadcase: Out of all Interface Entities, only those referenced by the Loadcase Header will be written in the *alc_aux* file.
Used by Assembly: Out of all Interface Entities, only those references by Connector Entities of entities that belong to Connecting Subsystems will be written in the *alc_aux* file.
Unused: Out of all Interface Entities, only those not referenced by any entities will be written in the *alc_aux* file.

A usual set-up for the Modular Environment is to write in the *alc_aux* file only those Interface Entities that are used by the Loadcase. This way, Interface Entities used for Assembly, are excluded, as they can be too many and may not be needed for post-processing. Such a set-up is shown in the screenshot below:

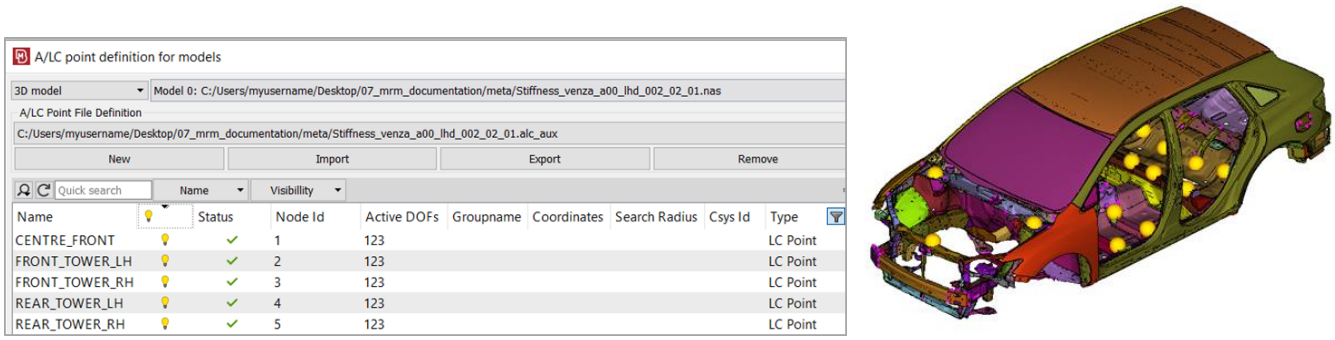


Note that in case of base modules (i.e Subsystems or Library Items) no filtering is applied and the *alc_aux* file which is produced contains information about all the included interface entities.

7.4.2. Reading alc_aux files in META

META attempts to detect automatically the *alc_aux* file that relates to the results being read based on the result file name. During the reading of the results (either 2D or 3D), if a file with the extension *.alc_aux* and the same name with the result file resides in the same folder, META will identify it and read it automatically. If no such file exists, any file with the extension *.alc_aux* will be read if found (suitable for d3plot result files for example).

Within META, the Interface Points are listed in the A/LC_Points list accessed through **Lists > A/LC_Points.**



Interface Points in META

This list displays the key characteristics of the Interface Points, like their name, the id of the node they mark, their type (A_POINTs or LC_POINTs), information on their local coordinate system and their active DOFs, information related to their grouping via Interface Sets, etc.

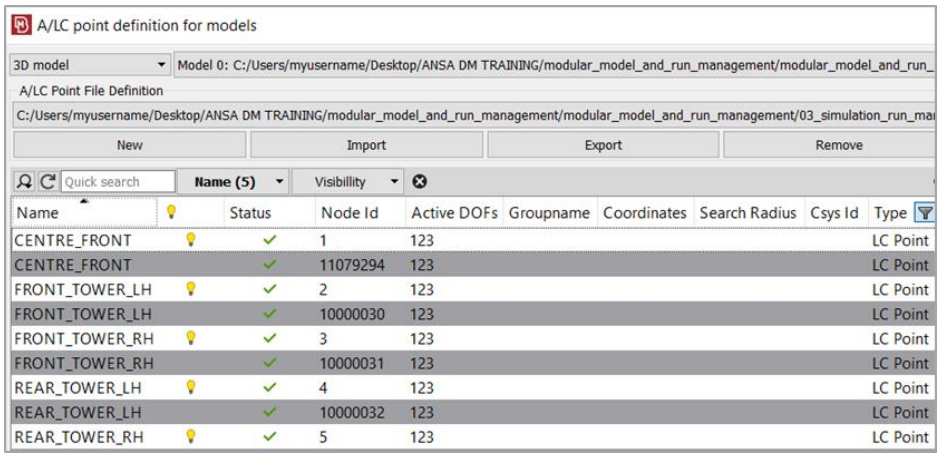
Through this window, it is possible to read individual *alc_aux* files using the **Import** function, as well as create new Interface Points using the **New** function.

Important notes!

1. Interface Points in META are identified by Name and therefore, Interface Points must have unique names. In case an *alc_aux* file contains more than one Interface Points with the same name that mark different nodes, a relevant message will be printed in the Info window and the last Interface Point definition will prevail:

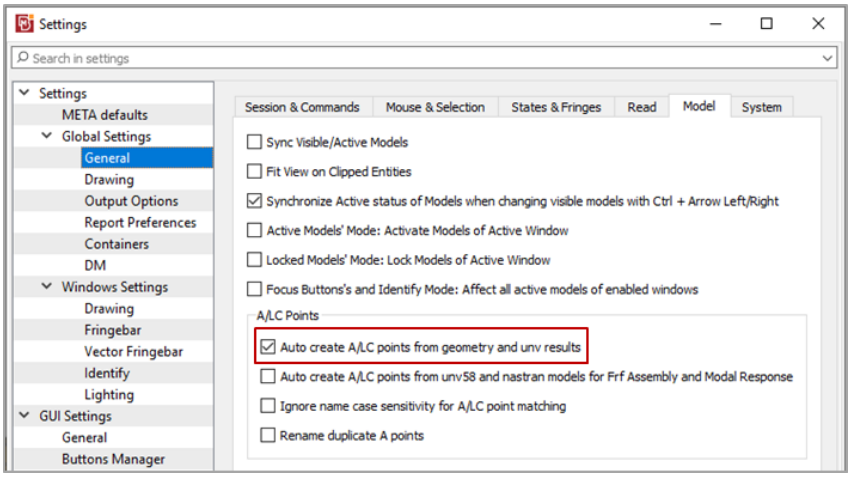
> *Warning! A/LC points with the same name have been found!*
> *The old A/LC points will be invalid*

Still, all Interface Point information will be available in the **A/LC_Points** list, but only one of the points with the same name will be drawn on the model and it's the node of this Interface Point that will be identified by the Interface Point name.



2. Even though one can distinguish Assembly Points from Loadcase Points through their type in the list, functionality-wise, there is no difference in the handling of the two. All functionality that is applicable to Loadcase Points is well applicable to Assembly Points too.

3. Interface Points in META can also be created based on the *field 10* marking of nodes in Nastran and based on the name info in Universal format files of datablock 58 or atfx files (files that usually hold measurements). The automatic creation of Interface Points based on this information can be activated through the option **Auto create A/LC points from geometry and unv results** found under *Global Settings > General [Model]*.



7.4.3. Using Interface Points in META

The primary use of Interface Points in META is for the easy identification of nodes. Using Interface Point names instead of the ids of the underlying nodes enables effortless comparison and combination of finite element results and experimental results.

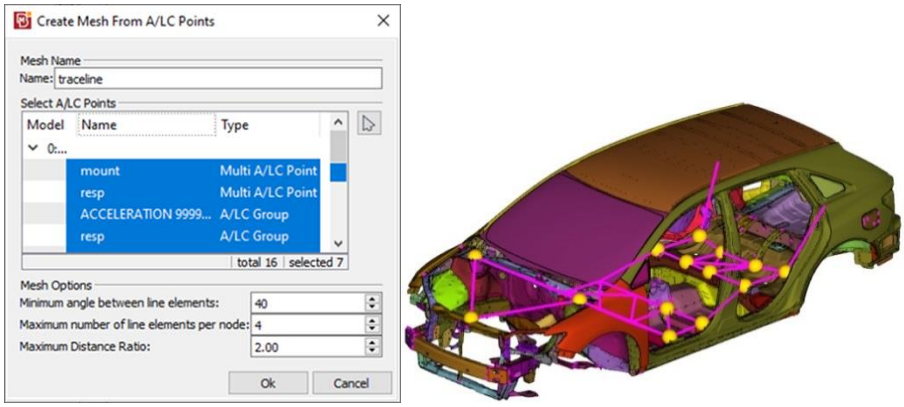
It is possible to instruct META to always use the name of the Interface Point of a node instead of its id in the recorded command in the session files, so that the recorded session is reusable on another model that has the same Interface Point names and possibly different underlying node ids. This behavior is controlled through the setting ***Session Output Rule: Names*** under *Global Settings > General [Session & Commands]*. In this way, the command of identifying a node will be written as:

identify node <***interface point name***>

Instead of:

identify node <***node id***>

A clound of Interface Points can be also used for the definition of a display model through the functionality accessible through the function **New > From A/LC Points** available in **Display Mesh** window.

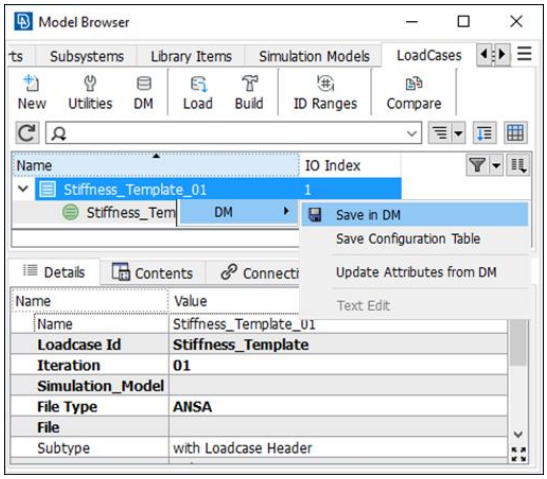


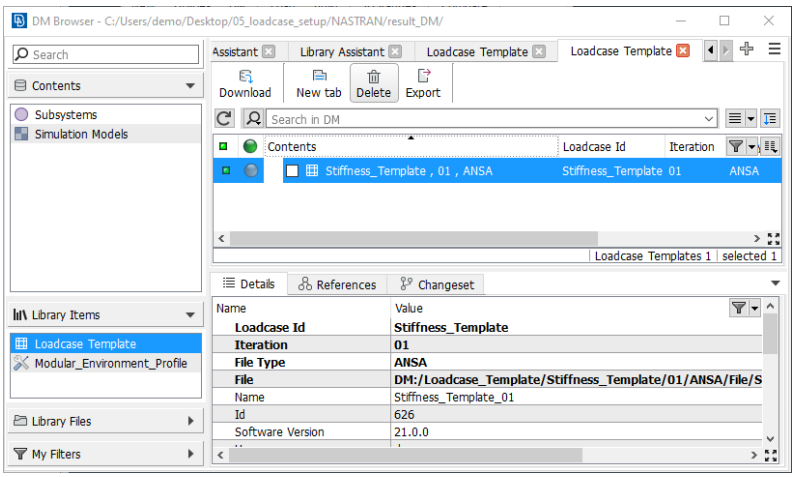
7.5. Saving Loadcases in DM

Loadcases can be saved in DM for different purposes:

Loadcase Template

A non-adapted Loadcase can be stored in DM as a template. In this case the Loadcase should be selected from the Loadcases tab in Model Browser. Non-adapted loadcases are stored as Loadcase Template library items in DM.

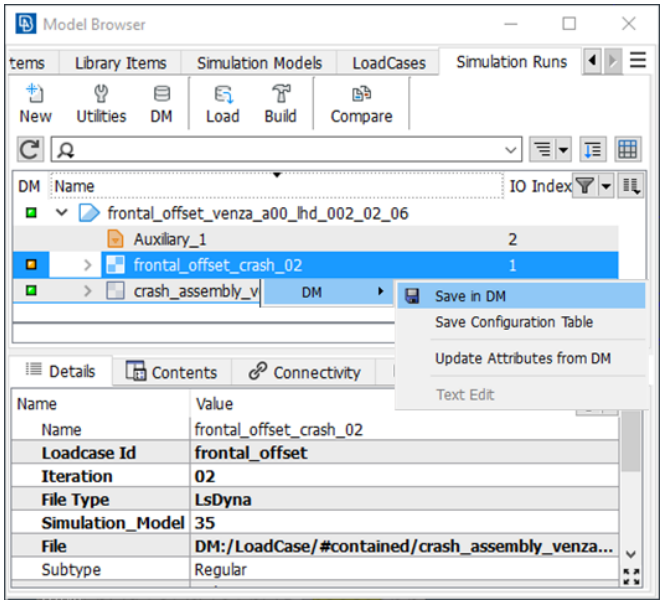


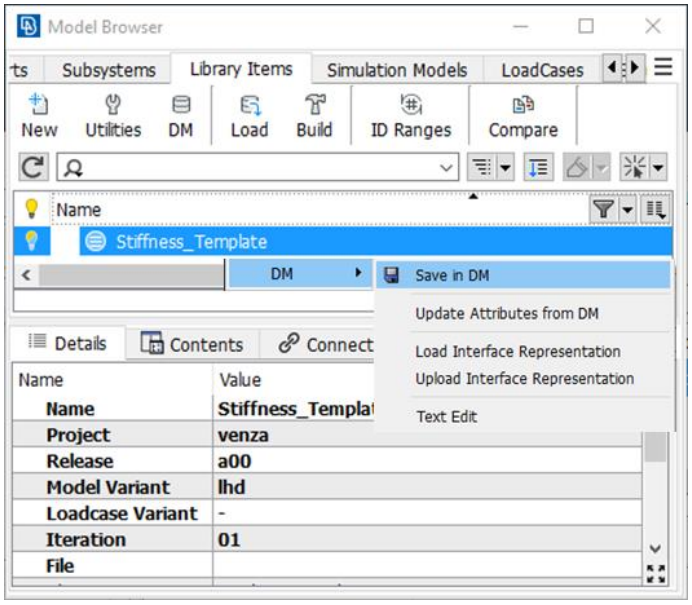


Adapted Loadcase

Depending on the set-up of the Simulation Run main file a Loadcase that is adapted for a particular Simulation Model may need to be stored in DM. In this case the Loadcase should be selected from the Simulation Runs tab.

Apart from saving in DM the Loadcase itself, the Loadcase specific Library Items (i.e. Loadcase Header, Target Points) can be stored in DM as templates, in order to be reused under different Loadcases.





8. Creating Simulation Loops

Creating new simulation loops in order to improve the behavior of the model, requires starting from a base Simulation Run (i.e. the run that must be used as a basis for the creation of the new iteration), making a modification to one or more of its contents and saving a new version of the Simulation Run that contains references to the new modified contents. This process in the Modular Environment is referred to with the term **Next Iteration**.

This chapter presents the recommended way of work in order to cover the following scenarios:

Scenario 1: Modify Subsystems standalone and update the related Simulation Runs

This scenario applies:

- In all cases where a single Subsystem needs to be modified
- In all cases where more than one Subsystems need to be modified, that are not *adapted* in their Simulation Models and Loadcases in a way that affects their position and ids

For the creation of simulation loops according to this scenario, the Subsystems are selected in the DM Browser, **Next Iteration** is performed to load them in ANSA, they are modified and then saved in DM as a new iteration.

Scenario 2: Modify Subsystems in the context of their parent Model Container and update related Simulation Runs

In the cases where Subsystems are *adapted* in their Simulation Models and Loadcases in a way that affects their position and ids and the new simulation loop needs to be made by modifying more than one Subsystems in the same session, processing the Subsystems standalone is not really possible since:

- In case id adaptation is used, the different Subsystems may have conflicting ids if loaded out-of-context (without their adapters)
- In case position adaptation is used, the different Subsystems may appear in the wrong position if loaded out-of-context (without their adapters)

In this scenario, adaptation rules need to be applied at the time of the modification.

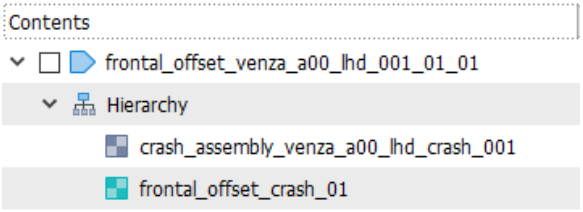
For the creation of simulation loops according to this scenario, the Simulation Model or Loadcase that contain the Subsystems to be modified are selected in the DM Browser, **Next Iteration** is performed to load them in ANSA and then the Subsystems are loaded from the Model Browser. This way, the Subsystems will be “adapted”, meaning that any positioning or renumbering will be applied on Load, they will be modified and then saved in DM as a new iteration. ANSA will automatically reverse the adaptations made when saving the Subsystems as a new iteration back in DM. For more details read chapter 5.4.

8.1. Scenario 1: Modify Subsystems standalone and update Runs

This is a two-step process. In the first step, new iterations of the Subsystems are created and saved in DM . In the second step, the Simulation Runs are updated with the new Subsystem versions and are stored as new Iterations in DM.

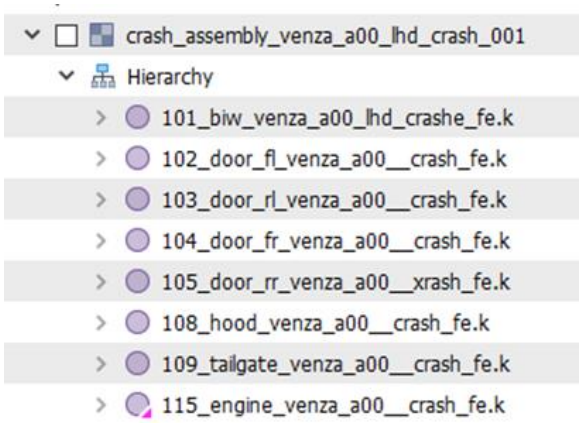
8.1.1. Create new Iterations of the Subsystems

8.1.1.1. New Iteration by modifying a Subsystem in ANSA

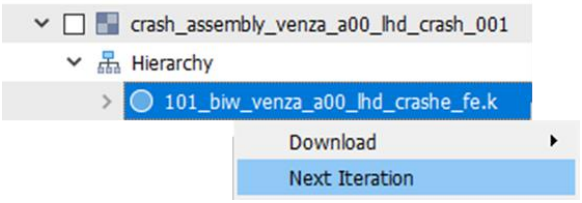


To identify the version of a subsystem used in a run, the Simulation Run of interest is found first in DM.

By expanding the Simulation Run contents in the DM Browser, the Simulation Model and Loadcase that are used are displayed.



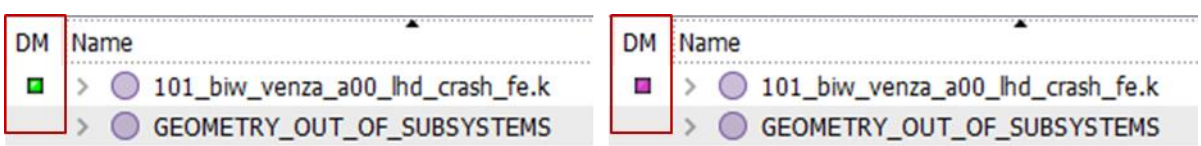
The next step would be to select the Simulation Model and click the **Open In New Tab option** available in the context menu. A tab will open in the DM Browser listing the Simulation Model. To see the contents of the Simulation Model, switch the list to the *Hierarchy* view mode. By expanding the Simulation Model, the hierarchy is opened and its contents are displayed.



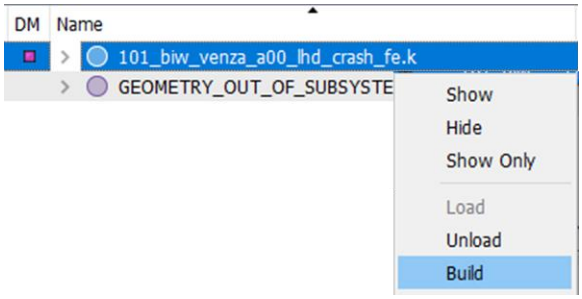
To create a new iteration of a Subsystem, select it and execute the **Next Iteration** command available in the context menu.

The selected Subsystem will be a solver keyword file. However, there's always a link between the solver format Subsystem product and the ANSA format Subsystem definition if the ANSA file is saved before the Solver file. So although the selected subsystem had a solver file type, the process will load in ANSA the definition file of that Subsystem, which has File Type = ANSA. Note that if for some reason the solver file type subsystem does not have a link to an ANSA subsystem, a Definition File Error will be shown.

After downloading the Subsystem the next step is to make a modification. Notice the subsystem's DM Update status. Once it is downloaded it is green meaning that it is "Up to date". After modifications take place it is updated and becomes "Modified" as seen in the image below.



In order to verify that a Subsystem is ready to be saved in DM, the Build action, that is available is the context menu, needs to be performed. This will realize any Connections, Connectors and GEB entities that exist in the Subsystem and finally will verify the Numbering Rules.

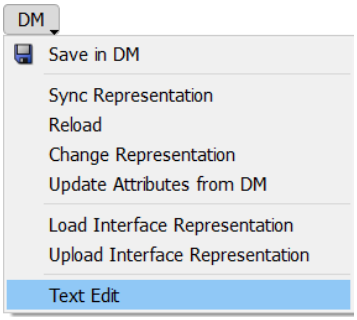


Once the Build process is successfully finished without any errors the Subsystem is ready to be saved in DM as a new Iteration.

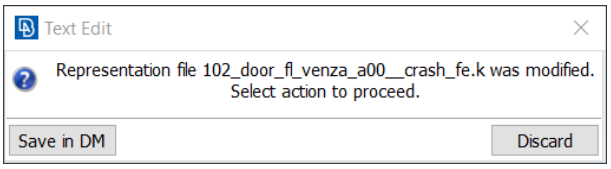
Note that the Build process can be automatically embedded in the Save Silently process, as described in **Chapter 3.7**

8.1.1.2. New Iteration by modifying a Subsystem in a Text Editor

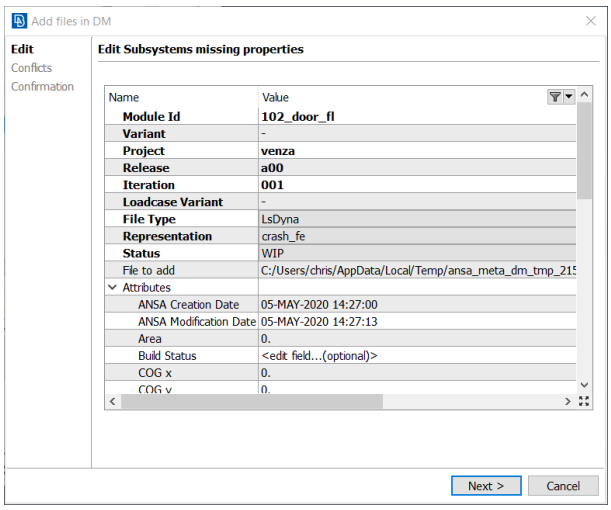
It is also possible to create a new Iteration of a solver type Subsystem by text editing the keyword file.



Once the solver file type subsystem definition is downloaded, the Subsystem keyword file can be viewed and edited from the **Text Edit** option available in the context menu of the Subsystem in Model Browser.



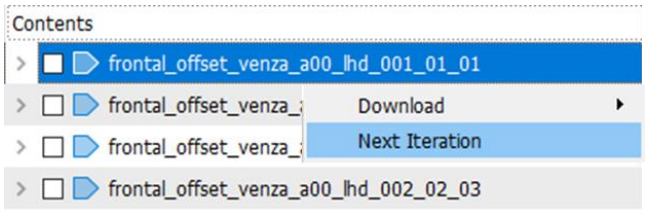
The default system text editor will be opened with the keyword file loaded. After modifying and saving the text file, ANSA will prompt the user to Save in DM the new file or Discard the changes.



On **Save in DM** option, the **Add files in DM** wizard will be displayed guiding the user to choose the properties of the Subsystem and resolve any Conflicts with already existing Subsystems with same properties in DM.

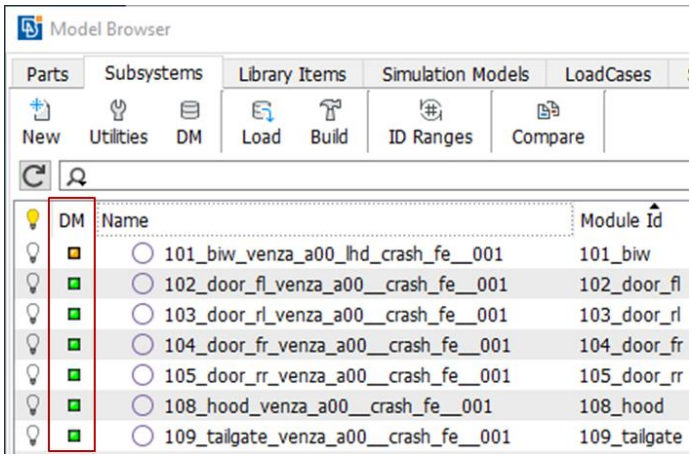
8.1.2. Create new Iterations on the Simulation Runs

In order to create new iterations of the simulation run, the base run is selected and the option **Next Iteration** is used to start the process.



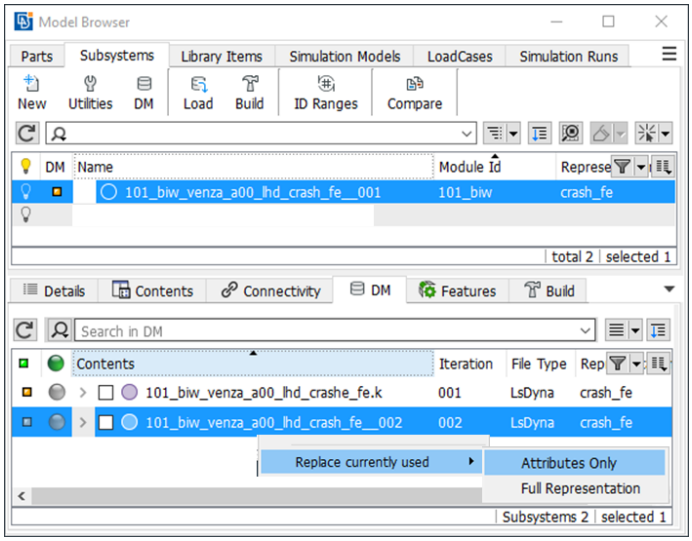
The **Next Iteration** option will load the run definition together with its required contents, so that the user can proceed to modifications.

Depending on the Save settings used to produce the run, the definition file of the run may contain only the model and loadcase or the model and the loadcase with their contents.

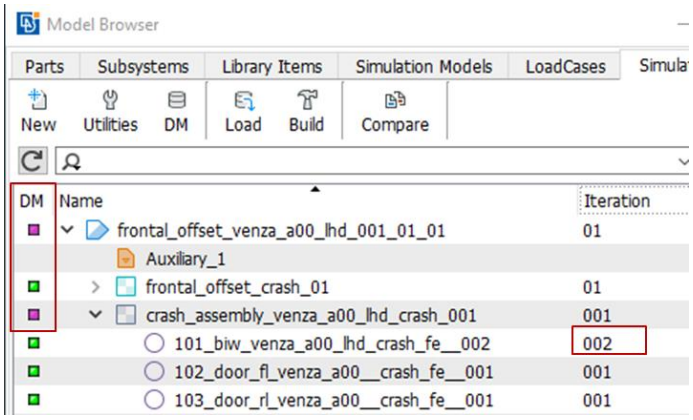


In case it contains only the model and the loadcase, the Next Iteration action will load the definition file of the run and the definition files of its contents sequentially (see paragraph 3.1.2)

Since new versions of the Subsystems exist, the DM Update Status is now orange, meaning that updates exist.

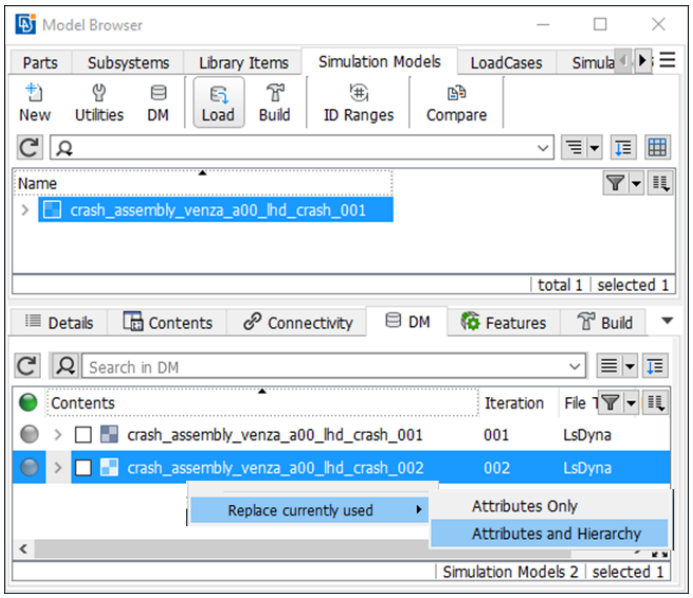


In the DM tab of the Model Browser the alternative versions of the selected Subsystem are listed. In order to update the Subsystem, the alternative version is selected and the **Replace currently used > Attributes only** option is selected.



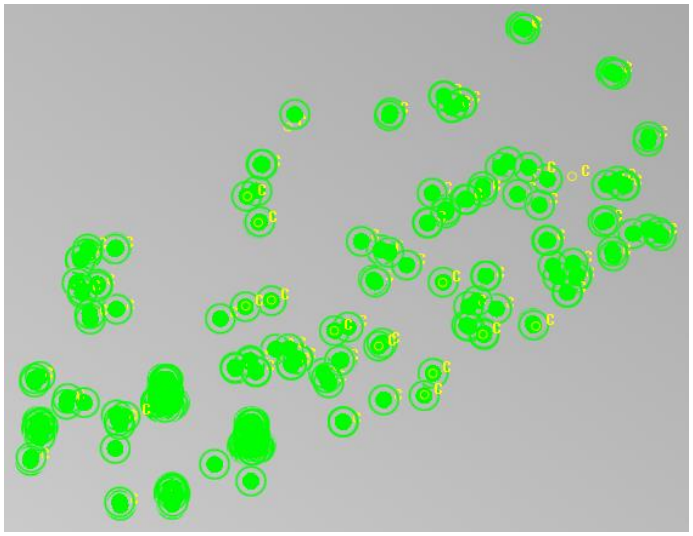
The attributes of the Subsystem will be replaced by those of the updated Subsystem. This modification is propagated and the DM Update Status of the Simulation Model and Simulation Run is changed.

Note that the definition attributes of the selected Subsystem (e.g. ID ranges, parameters), which are necessary to use the Subsystem in a compound container, are also automatically updated from DM.

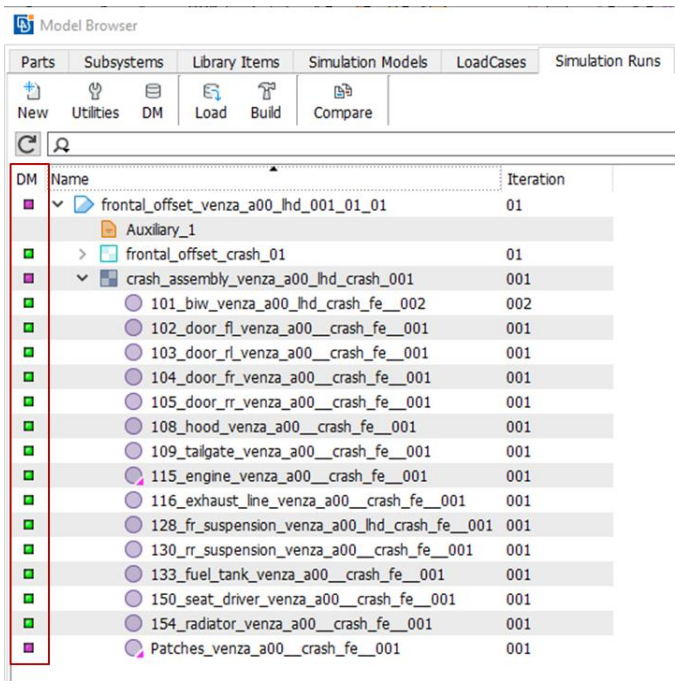


An alternative option is to update the Run by selecting a newer version of a compound container (i.e. Simulation Model or Loadcase). In this case the **Replace currently used > Attributes and Hierarchy** option should be selected. The function replaces not only the properties and attributes of the selected container but also its contents.

The next step is to execute the **Build** action on the Simulation Run. The **Build** action will be executed bottom-up, meaning that the lower level containers will be handled first moving on to the higher ones. In this step assembly takes place.

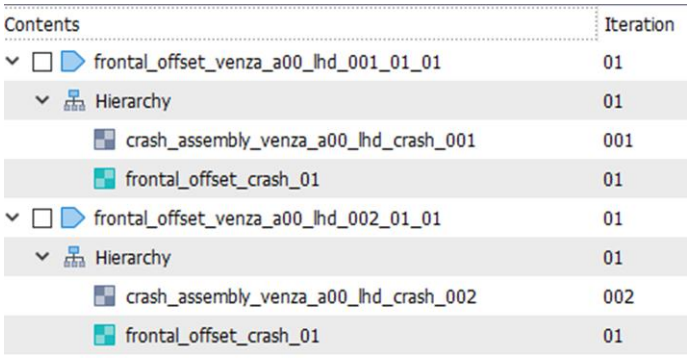


The interface representations of the Subsystems and Library items are loaded and the ANSA File type of the Connecting subsystems is loaded.



This is done in order to ensure the integrity of the model. For example, in case any of the interfaces have changed, the connecting subsystem is also modified and must be saved again.

Finally the Simulation Run needs to be saved in DM. By using the Silent Save functionality, the **Save in DM** action can be directly executed on the Simulation Run. Any missing or modified entities will be automatically saved based on the predefined Save Setup Settings.



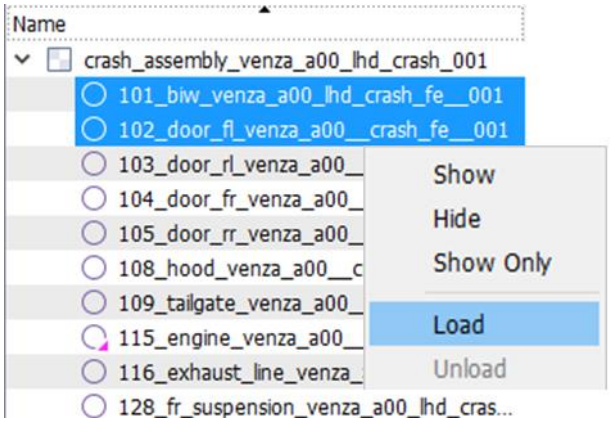
The DM now contains a new Simulation Run. The Run Iteration in both cases is 01, Loadcase iteration in both cases is 01 but the Simulation Model iteration was 001 and became 002.

Note that there is also a possibility to set the comment field mandatory in order to better track model changes through the ANSA.default settings:

- `make_comment_mandatory_for_save_iteration_of_subsystem`
- `make_comment_mandatory_for_save_iteration_of_simulation_model`
- `make_comment_mandatory_for_save_iteration_of_loadcase`
- `make_comment_mandatory_for_save_iteration_of_simulation_run`

8.2. Scenario 2: Modify Subsystems within their compound Model Container

In this case the Simulation Model or Loadcase that contain the Subsystems to be modified are selected in the DM Browser, and **Next Iteration** is performed. The definition file is loaded in ANSA. Then the next step is to load the Subsystems from the Model Browser.

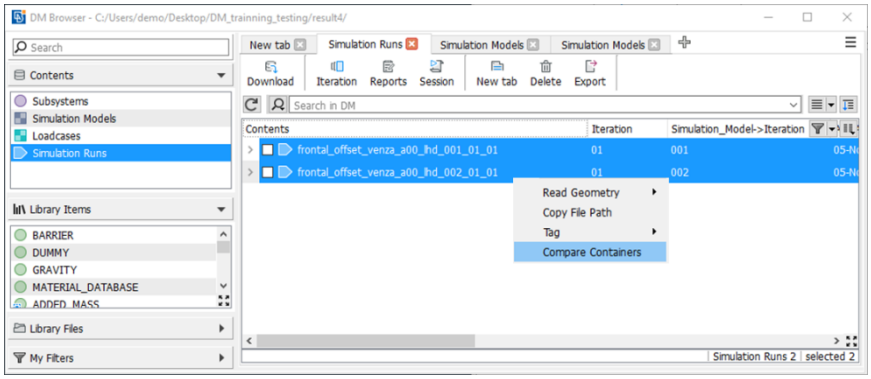


The result of loading the Subsystems from the Model Browser instead of downloading them from DM is that the Subsystems will be “adapted”, meaning that any positioning or renumbering will be applied on Load.

After modifications they are saved in DM as a new iteration.

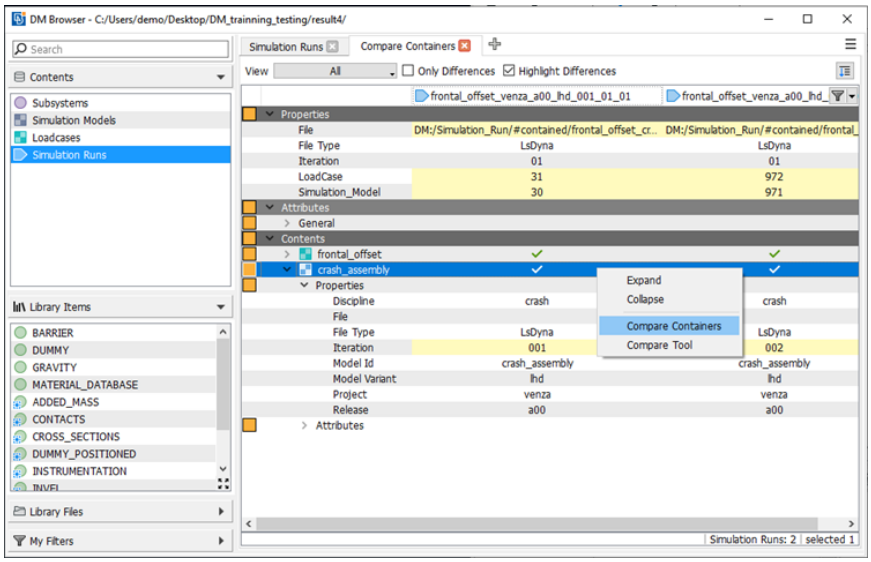
ANSA will automatically reverse any adaptation made when saving the Subsystems as a new iteration back in DM.

8.3. Compare MBContainers



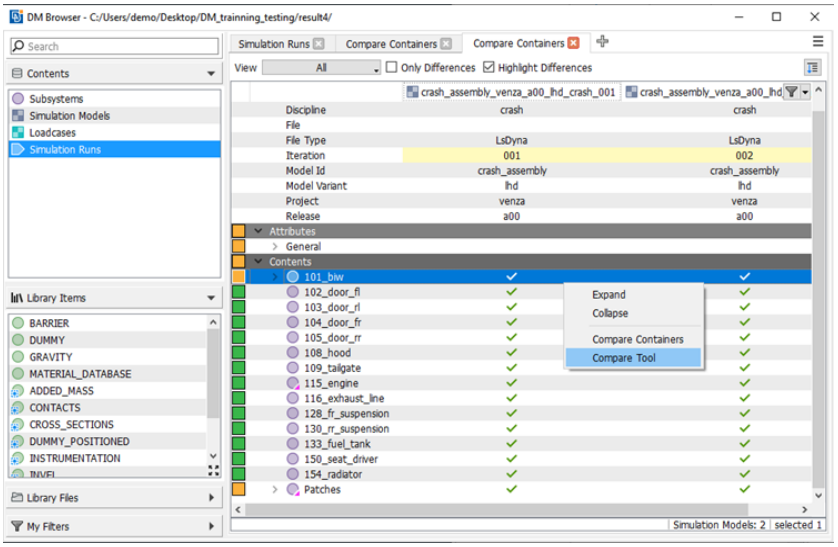
It is possible to compare the attributes and contents of Simulation Models, Loadcases and Simulation Runs in order to find any differences either in DM Browser or in Model Browser.

This can be achieved through the **Compare Containers** tool available in the context menu.



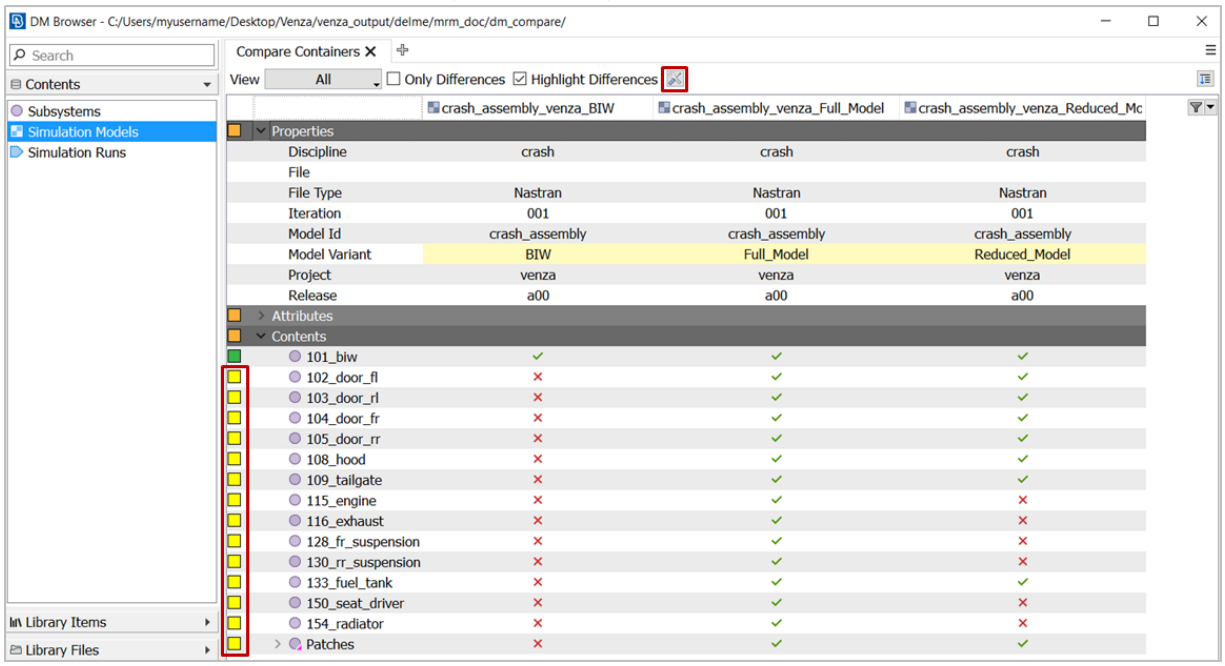
A new tab opens inside the DM Browser with the name “Compare Containers”. In this tab the selected objects are listed into separate columns. Their properties, attributes and contents are listed and any differences are highlighted.

The **Compare Containers** tool can be also launched for the contents of the containers. In order to do that, select the contents to compare, in this case the Simulation Model and select **Compare Containers** option from the context menu.

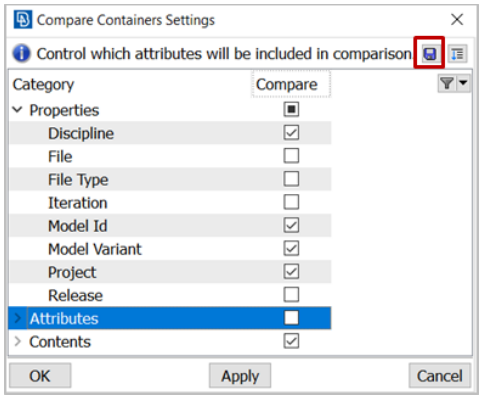


The comparison can proceed to a deeper level. The different subsystems can be selected and this time the **Compare Tool** option of the context menu can be activated to perform a geometric comparison of the two subsystems in ANSA.

In cases where more than two containers are compared and some of the contents do not participate in all of them, they will be marked as yellow. This indication is used in order to distinguish them from contents that do participate in all containers but have differences (Orange color).



A "*Settings*" button allows the user to determine which attributes will be compared. Entire categories of attributes can be excluded from the comparison process in a single shot. Moreover, the selections can be saved in ANSA.defaults.



9. Management of key-results and reports

9.1. Results handling in the Modular Environment

The Modular Environment offers a complete solution for the management of simulation results. The key characteristics of this solution are listed below:

- Processed results are stored in the data repository under the Simulation Run they concern
- In ANSA and META, the main user interface for the management of the results is the **DM Browser** which comes with the following functionality dedicated to results assessment:
  - **Reports Table**, for the comprehensive presentation of the results of selected simulations in a per-type fashion
  - **Embedded META Viewer**, for the visualization of processed results of all types (values, 2d-plots, 3d model) with integrated automatic "overlay" functionality for the effortless comparison of results produced by different Simulation Run versions
  - **Results Logbook**, for the tabulated presentation of key-values that enables quick comparison of the results of different simulations and simultaneous conditional formatting for the quick visual comparison of results against threshold values
  - **Run composition comparison tool**, for the comparison of two or more versions of Simulation Runs in terms of Simulation Model and Loadcase composition, in order to easily identify the changes that led to differences of the results
  - **Post-processing launcher**: It can launch META in order to run a session file coming from the library on the "raw" results of a Simulation Run
- The same functionality is available in a similar fashion through KOMVOS
- Results handling is possible through both ANSA and META, providing the exact same functionality to the analyst. This way, a user in ANSA doesn't need to move to META in order to review the results of a past simulation.
- It is possible to associate results to Loadcases and Simulation Models directly, in order to facilitate the storage of reports that accumulate results coming from different Simulation Runs under the Loadcase (e.g. pedestrian protection simulations) or results that combine different Loadcases under the Simulation Model.
- The "raw" results of the solver do not need to be stored within the data repository

An extensive description of the various tools for the handling of simulation results is given in Chapter 4 of the Data Management Reference Guide.

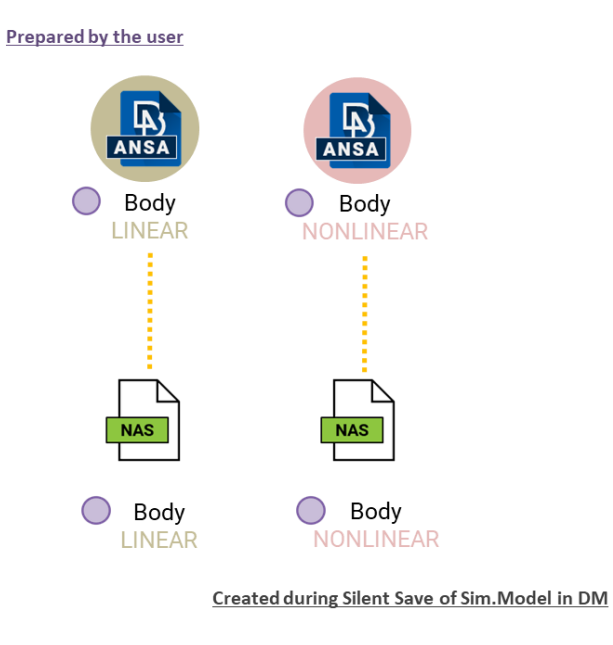
Special topics

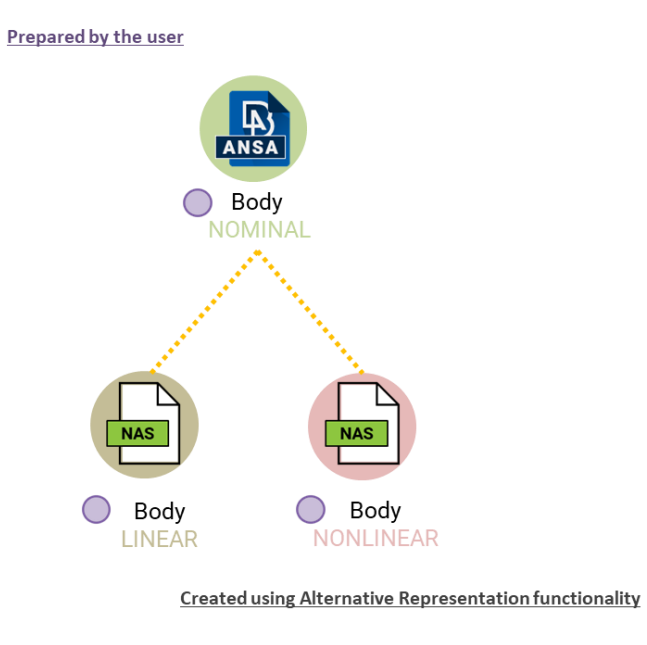
10.1. Alternative FE Representations

With the default set-up in the Modular Environment, each Subsystem is stored as an ANSA file and the solver keyword file is generated “just in time”, when it is required for the setup of a compound Model Container. This way, there’s a one-to-one relationship between the solver keyword file and the ANSA file that was used to generate it. The two objects in DM have identical primary attributes except for their File Type. Their relationship is captured as a “derivation” link between the respective DM objects so that every time the user needs to make modifications and create new versions of the Subsystem, the system suggests the ANSA file as the *base* representation.

However, there are cases where different Loadcases require slightly different variations of a Subsystem that can all be generated based on the same ANSA file with minimal changes in the Build Process. In these cases, the default handling of the Modular Environment would require the maintenance of several ANSA Subsystems, that would need to evolve synchronously. Such handling would be cumbersome and would lead to a waste of storage space. For these cases, the Modular Environment supports the concept of *Alternative FE Representations* that enables the same ANSA file to be used for the derivation of several solver Subsystems. With this concept, there’s a one-to-many relationship between the ANSA and the solver Subsystems. The different objects in DM will have identical primary attributes except for their File Type and the Loadcase Variant (or any other primary attribute used to capture the loadcase variation). A couple of examples where the concept of *Alternative FE Representations* can be applied are given below:

- In safety simulations, a seat Subsystem that is positioned differently for different Loadcases.
- In durability simulations, a Subsystem whose materials have linear behavior for some Loadcases and non-linear for others.



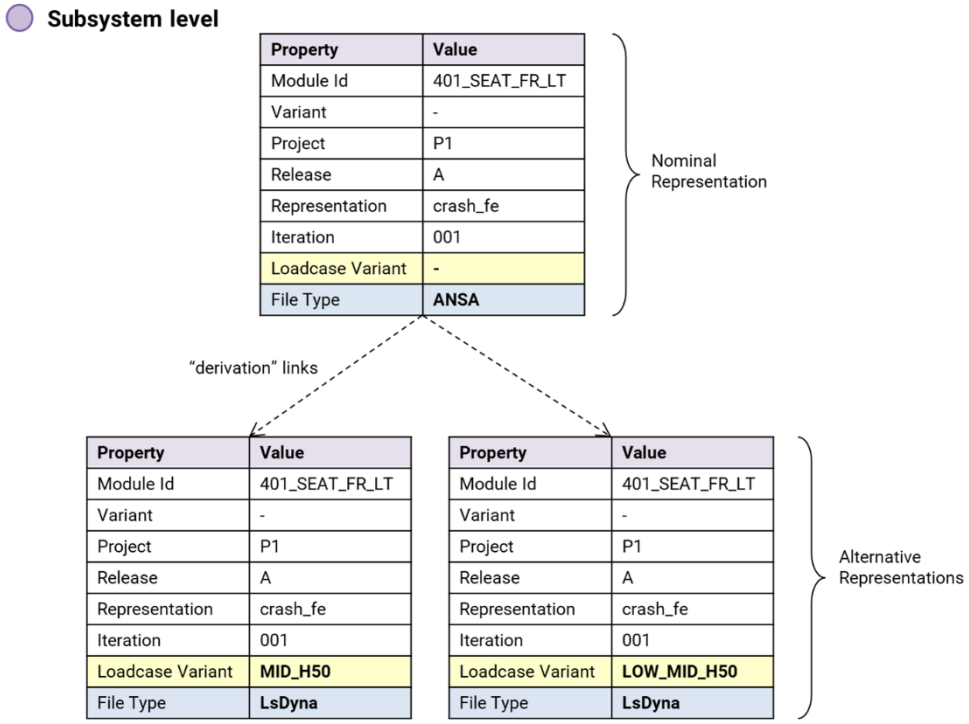


10.1.1. Terminology

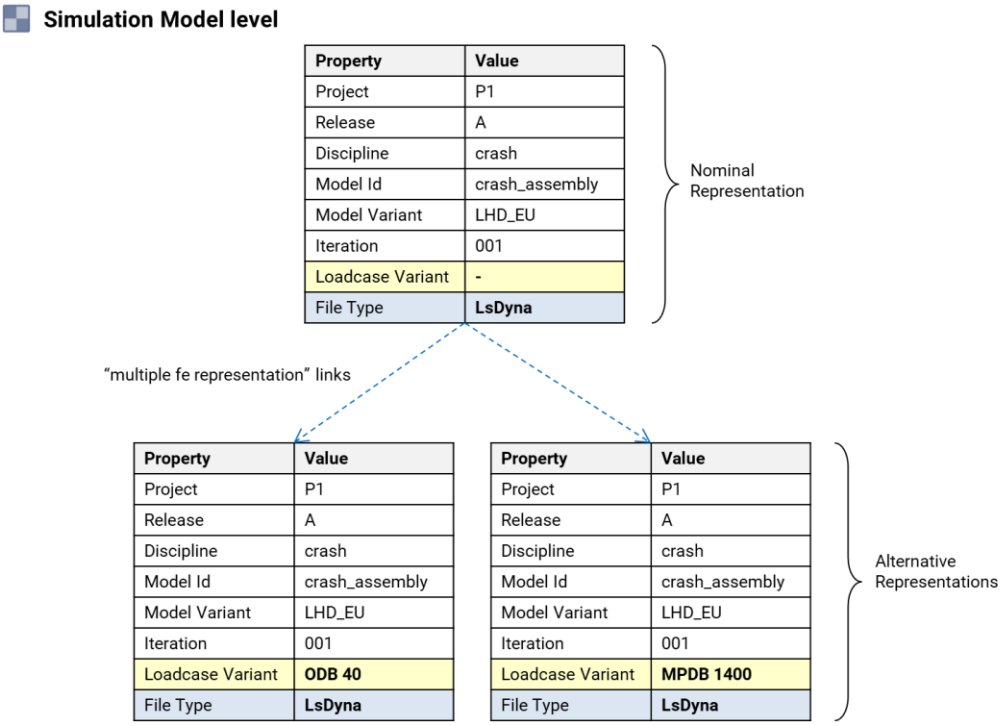
Let's take as an example the seat Subsystem in two different safety Loadcases, a 40% Offset Deformable Barrier (ODB 40) and a Mobile Progressive Deformable Barrier of 1400Kg (MPDB 1400). The seat must be positioned in two different positions respectively, as shown in the table below:

Loadcase	Seat position
ODB 40	MID_H50
MPDB 1400	LOW_MID_H50

Let's assume that the information of the loadcase-driven variation (i.e. seat position) is kept in the Loadcase Variant primary attribute of the Subsystem. The two Loadcase Variants are created from the same ANSA Subsystem. In the Modular Environment, the ANSA Subsystem is called the *Nominal representation* and the two Loadcase Variants are called *Alternative representations.* Again, the ANSA Subsystem is connected to the solver Subsystems with derivation links.



Going one level higher in the modular structure, the use of a different Loadcase Variant of the seat Subsystem triggers the creation of different Loadcase Variants of the Simulation Models too. The terms *Nominal* and *Alternative Representations* are also applicable here.



10.1.2. Preconditions

The *multiple FE-representations* concept is only applicable when the following pre-conditions are met:

- The assembly of all *alternative* representations to the rest of the model is identical to that of the *nominal* representation. The creation of a Loadcase Variant from the nominal representation should not affect the interfaces of the Subsystem.
- The derivation of a Loadcase Variant from the nominal representation can be implemented with a user-script Build Action. In the example with the seat Subsystem above, this Build Action would apply a certain Kinetic Position of the seat kinematic mechanism.
- The derivation of the Loadcase Variants from the nominal representation can take place in batch, with no user interaction

10.1.3. Use-cases covered

The table below presents the three main use-cases facilitated by the *Alternative FE Representations* functionality:

Standard MRM functionality	Extra capabilities with Alternative FE Representations
Use-case 1: Identify and set proper Loadcase Variant based on the Loadcase in hand	
For a user to set the right Loadcase Variant to Subsystems when their Simulation Model is combined with different Loadcases, every time a Run is set-up and the Loadcase is defined, the user should manually update the Loadcase Variant of the Subsystems and the Simulation Model	The right Loadcase Variant is set to the Subsystems and the Simulation Model automatically during Build.
Use-case 2: Generate solver file of a particular Loadcase Variant of a Subsystem	
For a user to generate the solver file of a particular Loadcase Variant of a Subsystem, there should be a different ANSA file for each different Loadcase Variant	The solver file of a particular Loadcase Variant of a Subsystem can be generated automatically by executing the Subsystem's Build process on the Nominal Representation (ANSA file).
Use-case 3: Auto-generate solver file of a particular Loadcase Variant of a Subsystem during the Saving of a Simulation Model as solver file with references	
For the system to auto-generate the solver file of a particular Loadcase Variant of a Subsystem, the ANSA file of that particular Loadcase Variant should already exist in DM.	The solver file of any Loadcase Variant can be auto-generated by the system based on the Nominal Representation (ANSA file) after executing the Subsystem's Build process in batch.

The implementation-specifics of each use-case are described below.

Use-case 1: Identify and set Loadcase Variant based on the Loadcase in hand

For the system to detect and set the proper Loadcase Variant during Build, a script function is used to decide the target Loadcase Variant in each case. This function is called automatically by ANSA the moment the user triggers Build (just before the built-in Build workflow starts) and for all those Model Containers that have an attribute declared for the control of alternative FE representations.

This use-case requires the following configuration:

1. In the *dm_structure.xml*:
   - Declaration of the Subsystem and Simulation Model primary attribute that will capture the loadcase variation (e.g. Loadcase Variant)
   - Declaration of the possible values of this primary attribute (e.g. "-", "ODB 40", "MPDB 1400")
   - Declaration of the Nominal and Alternative values of the Loadcase Variant (e.g. Nominal Value is the dash "-" and Alternative Values are "ODB 40", "MPDB 1400")
2. In the *ANSA.defaults*:
   - Definition of the script file and script function that will be used to decide the proper Loadcase Variant during Build

More information and examples for the configuration are given in paragraph 10.1.4.

Once this set-up is in place, during Build of any Model Container ANSA will call the designated script function, sending the following information as input arguments:

- **mbc_build_target**: The ANSA entity of the build target Model Container, i.e. the Model Container that is about to be built. Remember than when Build is triggered on the Simulation Run, the Build Processes of all contained Model Containers are executed bottom-up.
- **attr_name**: The name of the primary attribute used to capture the loadcase variations ("DM/Loadcase Variant" in the seat example above). As described above, this attribute is declared in the dm_structure.xml.
- **exec_args**: A dictionary with additional information about the process. For the time being, it only contains the key "PROCESS_EXECUTOR" that has as value the ANSA entity of the Model Container on which Build was requested. For example, if Build is triggered on the Simulation Run but, while the process is running bottom up, the function is called for a Subsystem, the Simulation Run entity will be given as PROCESS_EXECUTOR and the Subsystem as *mbc_build_target*.

From this script, ANSA expects a signal on how to proceed. Three signals are currently supported:

- If the loadcase variant value was successfully retrieved and must be set to the **mbc_build_target** before ANSA continues to the standard Build Process, the following dictionary must be returned:

  ```
  {'Action':'Assign', 'Value': <loadcase variant name>}
  ```

- If the **mbc_build_target** is not eligible for loadcase-base variation (e.g. if it's the body subsystem and not the driver's seat) ANSA should continue with the execution of the standard Build Process, the following dictionary must be returned:

  ```
  {'Action:'Pass'}
  ```

- If the script detects an error and ANSA should not continue with the execution of the standard Build Process, the following dictionary must be returned:

  ```
  {'Action':'Abort'}
  ```

Let's see use-case 1 in the example with the seat Subsystem and let's assume that depending on the *Loadcase Id* primary attribute of the Loadcases, the Loadcase Variant of Subsystems and Simulation Models must be set according to the table below:

Loadcase Id	Subsystem Loadcase Variant	Simulation Model Loadcase Variant
ODB 40	MID_H50	ODB 40
MPDB 1400	LOW_MID_H50	MPDB 1400

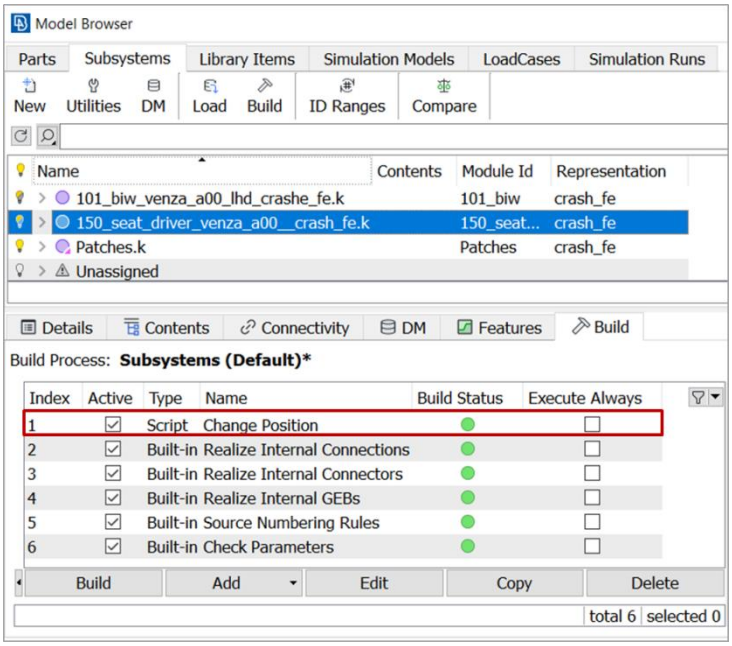
A sample script for the seat example is given below.

Pre-Build script to decide Loadcase Variant

```
def get_loadcase_variant_info (mbc_build_target, attr_name, exec_args):

    mbc_build_source = exec_args['PROCESS_EXECUTOR']
    type_of_mbc_build_target = mbc_build_target.ansa_type(0)
    type_of_mbc_build_source = mbc_build_source.ansa_type(0)


    if
_is_sm_eligible_for_alternative_representation(mbc_build_target,type_of_mbc_build_target):
        return {'Action' : 'Pass'}
    if
_is_sub_eligible_for_alternative_representation(mbc_build_target,type_of_mbc_build_target):
        return {'Action' : 'Pass'}

    current_lc_variant = mbc_build_target.get_entity_values(0, [attr_name])[attr_name]
    target_lc_variant = None

    if type_of_mbc_build_source == 'ANSA_SIMULATION_RUN':
        lc_of_run = _get_loadcase_of_sim_run (mbc_build_source)
        target_lc_variant = _get_loadcase_variant_from_lc_ent (lc_of_run,
type_of_mbc_build_target)
    elif type_of_mbc_build_source == 'ANSA_LOADCASE':
        target_lc_variant = _get_loadcase_variant_from_lc_ent (mbc_build_source,
type_of_mbc_build_target)
    elif type_of_mbc_build_source == 'ANSA_SIMULATION_MODEL':
        target_lc_variant = _get_loadcase_variant_from_sm_ent
(mbc_build_source,type_of_mbc_build_target)

    if target_lc_variant is None and
_is_current_lc_representation_the_nominal(mbc_build_target, attr_name):
        print('Alternative Representation: No Loadcase Variant to assign and container has
nominal representation')
        return {'Action' : 'Pass'}
    elif target_lc_variant == current_lc_variant:
        return {'Action':'Pass'}
    elif type_of_mbc_build_source != "ANSA_SIMULATION_RUN":
        pass
        print('Alternative Representation: Executor is not the Simulation Run')
        return {'Action' : 'Pass'}
    elif _is_loaded_Subsystem(mbc_build_target) and not
_is_current_lc_representation_the_nominal(mbc_build_target, attr_name):
        pass
        print('Alternative Representation: Subsystem is not the nominal')
        return {'Action' : 'Pass'}
    elif target_lc_variant:
        print('Alternative Representation: Assign: ' + target_lc_variant)
        return {'Action' : 'Assign', 'Value' : target_lc_variant}
    else:
        return {'Action' : 'Pass'}

    return target_lc_variant
```

Use-case 2: Generate solver file of a particular Loadcase Variant of a Subsystem

For the system to generate automatically a particular Loadcase Variant of a Subsystem based on the Nominal Representation (ANSA file), the Subsystem's Build Process is customized by adding one or more script Build Actions. These Build Actions will decide how to modify the Subsystem based on the current value of the Loadcase Variant primary attribute.



A sample script Build Action for the seat example is given below:

Build Action on Subsystem to create Loadcase Variants from Nominal Representation

```
def main(subsystem):

    ret_eligible=_eligibleForBuild(subsystem)
    if ret_eligible:
        return ret_eligible
    else:
        ret = position_seat_build_action(subsystem)
        return ret

def position_seat_build_action (subsystem):

    deck = base.CurrentDeck()
    lc_variant = subsystem.get_entity_values(deck, ['DM/Loadcase Variant'])['DM/Loadcase
Variant']
    if lc_variant in ['MID_H50','LOW_MID_H50']:
        kin_pos = _get_kinetic_position_by_name (lc_variant)
        if not kin_pos:
            print ('Kinetic Position with name {} does not exist'.format(lc_variant))
            return False
    elif lc_variant == '-':
        print ('No need to change kinetic position of the Subsystem')
        return True
    else:
        print ('Unknown Loadcase Variant {}'.format(lc_variant))
        return False
```

```
    is_pos_ok = kinetics.MoveToPosition(kin_pos)
    if is_pos_ok:
        return True
    else:
        print ('Error during positioning')

    return False
```

This way, a user can derive the different alternative representations of a Subsystem by executing the following steps:

1. Load Nominal Representation of Subsystem from DM
2. Edit Loadcase Variant primary attribute
3. Build
4. Save in DM

Use-case 3: Auto-generate solver file of a particular Loadcase Variant of a Subsystem during the Saving of a Simulation Model as solver file with references

For the system to automatically generate the solver file of any Loadcase Variant based on the Nominal Representation (ANSA file), the Subsystem's (Nominal Representation) Build Process must include the Build Action(s) described in use-case 2.

This way, a user can create the solver files of the different alternative representations "just in time", at the time the Simulation Model solver file is saved in DM, with the following process:

1. Build Simulation Model, which ends-up to the setup of the proper Loadcase Variant to the Subsystems and the Simulation Model
2. Save Simulation Model as solver file with references
3. Run integrity checks for Simulation Model, detect that the solver representation of the requested Loadcase Variant does not exist for a Subsystem and trigger "auto-fix":
    a. Auto-load the Nominal Representation of the Subsystem in the background
    b. Set Loadcase Variant value
    c. Execute Build Process of Subsystem
    d. Save solver file of Subsystem

10.1.4. Configuration

The *Alternative Representations* require the following configuration:

In the dm_structure.xml:

A declaration in the **dm_structure.xml** of which property will be used to determine the Alternative Representations of the Subsystem and the Simulation Model. This declaration is made with the *"multiple_fe_representation_values"* keyword in the definition block of the property to be used. This keyword gets as value the comma-separated list of alternative representations that must always be a subset of the values defined in the "accepted_values" keyword.

Note ! The first value of the *"multiple_fe_representation_values"* keyword (e.g "-") will be considered by ANSA as the Nominal representation.

```
<Subsystem>
  <Properties>
    <Property name="Module Id" type="TEXT" allow_null="NO" read_only="NO"/>
    <Property name="Variant" type="TEXT" default_value="-" allow_null="NO" read_only="NO"/>
    <Property name="Project" type="TEXT" default_value="-" allow_null="NO" read_only="NO"/>
    <Property name="Release" type="TEXT" default_value="-" allow_null="NO" read_only="NO"/>
    <Property name="Iteration" type="VERSIONING SCHEME COUNTER" default_value="001" allow_null="NO" read_only="NO" format="%03d"/>
    <Property name="Loadcase Variant" type="TEXT" accepted_values="-,MID_H50,LOW_MID_H50" default_value="-" allow_null="NO" multiple_fe_representation_values="-,MID_H50,LOW_MID_H50" read_only="NO"/>
  </Properties>
</Subsystem>
```

```
<Simulation Model>
  <Properties>
    <Property name="Discipline" type="TEXT" default_value="durability" accepted_values="crash,nvh,durability,cfd" allow_null="YES" read_only="NO"/>
    <Property name="Model Id" type="TEXT" default_value="crash_assembly" accepted_values="crash_assembly,pedestrian_assembly,body,seat_dummy" allow_null="NO" read_only="NO"/>
    <Property name="Model Variant" type="TEXT" default_value="-" allow_null="NO" read_only="NO"/>
    <Property name="Project" type="TEXT" default_value="-" allow_null="NO" read_only="NO"/>
    <Property name="Release" type="TEXT" default_value="-" allow_null="NO" read_only="NO"/>
    <Property name="Iteration" type="VERSIONING SCHEME COUNTER" default_value="001" allow_null="NO" read_only="NO" format="%03d"/>
    <Property name="Loadcase Variant" type="TEXT" accepted_values="-,ODB 40,MPDB 1400" default_value="-" allow_null="NO" multiple_fe_representation_values="-,ODB 40,MPDB 1400" read_only="NO"/>
</Properties>
</Simulation Model>
```

In the ANSA.defaults

ScriptFunctionToObtainValueOfMultipleFeRepresentationAttribute
This setting controls which function will be triggered during the "Build" process of the Model Browser containers. In particular, the function based on the value of the "Loadcase Id" property of the Loadcase assigns the appropriate Alternative Representation to the Simulation Model and desired Subsystem containers. Moreover, performs several checks to ensure that an Alternative Representation will be created only for eligible Model Browser containers.

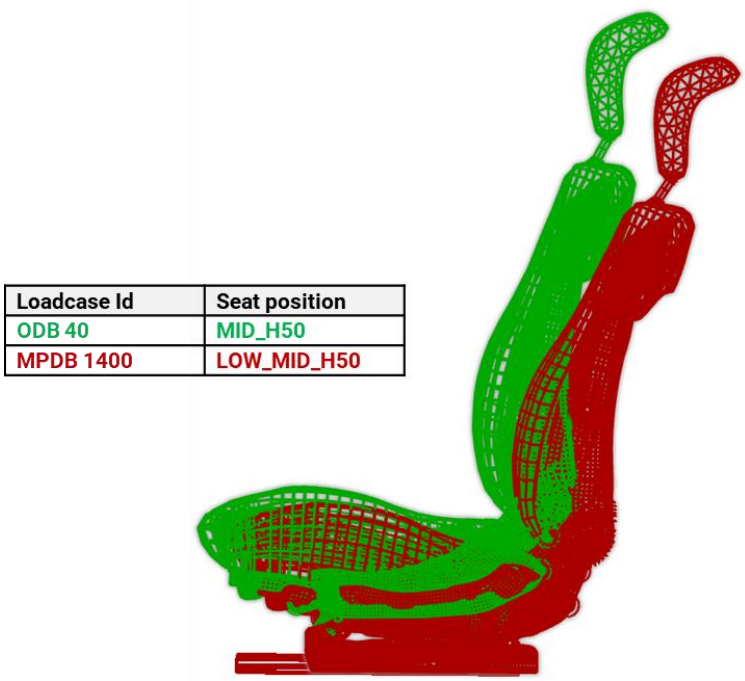
In the Modular Environment profile

A user script Build Action that needs to be added in the default Build Process of the Subsystem. This custom Build Action is responsible for the generation of an Alternative Representation based on the Nominal Ansa one.

This functionality is described in paragraph 10.1.2, while how to add a custom Build action is described in paragraph 4.2.3.

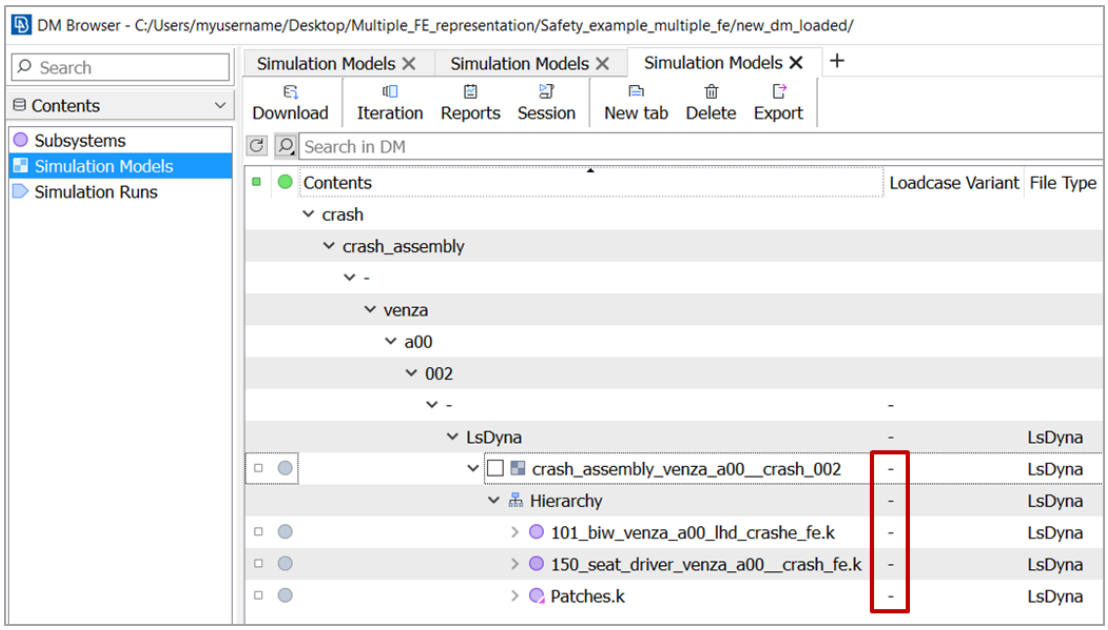
10.1.5. Working with Alternative FE Representations

As mentioned in paragraph 10.1, safety simulations are a typical example of a use case where Alternative Representations functionality can be applied. Let us examine the case where a Seat Subsystem will be placed in different positions to facilitate the set-up of two different safety Loadcases, a 40% Offset Deformable Barrier (ODB 40) and a Mobile Progressive Deformable Barrier of 1400Kg (MPDB 1400). The seat must be positioned in two different positions which will be distinguished in the model by using a different value on the “Loadcase Id” property of the Loadcase container, as shown in the screenshot below.



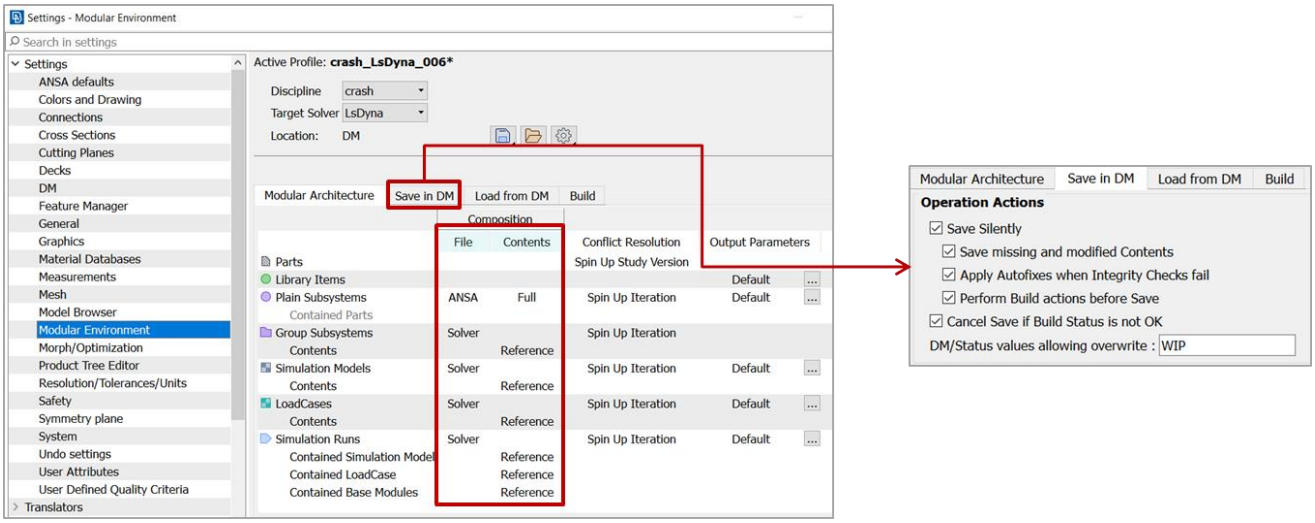
The information regarding the seat position is stored in the Loadcase Variant property of the Subsystem, while the information regarding the Loadcase setup is stored in the Loadcase Variant of the Simulation Model. Using the Alternative Representation approach, a single ANSA Subsystem will be used as a basis to derive the two Loadcase Variants of the Seat Subsystem. This Subsystem in the Modular Environment is called the Nominal representation and the derived Subsystems are called Alternative Representations. The Nominal representation is saved in DM as an ANSA file, while the Alternative Representations as solver keyword files. These different representations are linked to each other with representation derivation links. Accordingly, there is a Nominal Simulation Model and Alternative ones, which are again connected with Alternative Representation links.

Before proceeding to the generation of Alternative Representations it is necessary to ensure that the Nominal Simulation Model and Subsystem already exist in DM, as shown in the screenshot below.



In the Modular Environment settings the following file composition and format should be set in order to ensure an error prone process:

- The “Silent Save” option will be used in order to save using the recommended pre-configured “save scenario” of Modular Environment.
- “References” option was selected regarding the contents of higher level Model Browser containers.

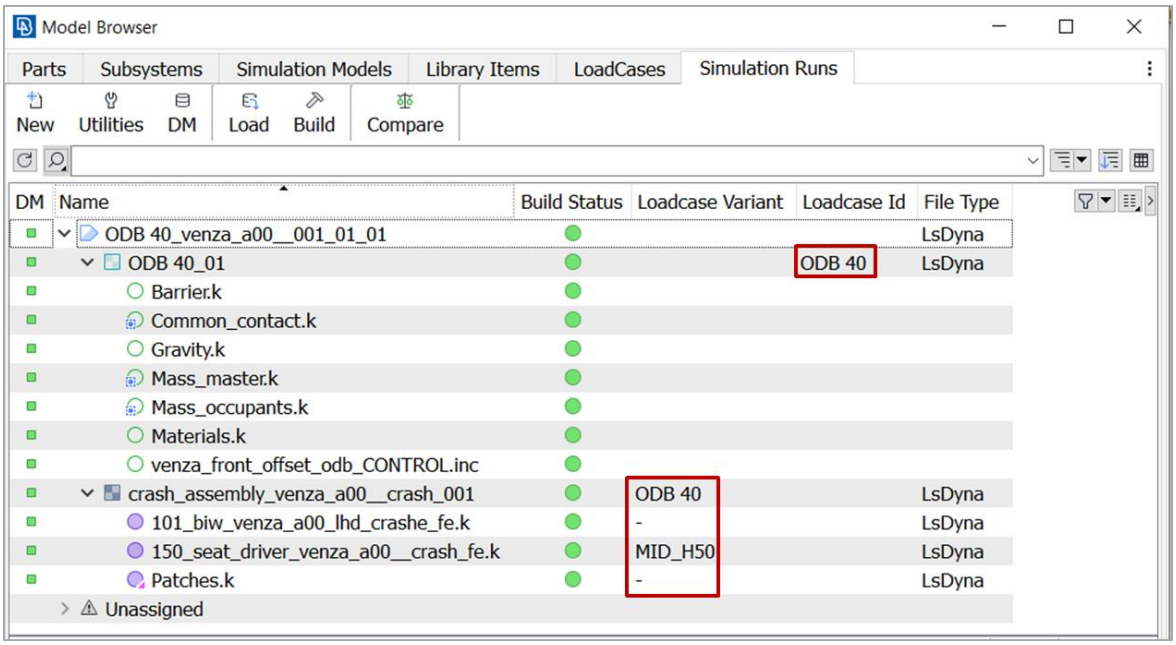


The first step would be to set-up the safety Loadcases, after that several Alternative Representations of the Simulation Run can be created as follows:

- Perform Next Iteration action on the Nominal Simulation Model.
- Set the proper value to the Loadcase property that drives the selection of the Alternative Representation for Subsystems and Simulation Models, in accordance to what’s expected by the script function declared in the ANSA.defaults. In this case, it’s the “Loadcase Id” property.
- Using the created Loadcase and the Nominal Simulation Model, create a new Simulation Run and save it in DM.

Note! The "-" value on the "Loadcase Id" and "Loadcase Variant" properties represents the Nominal representation.

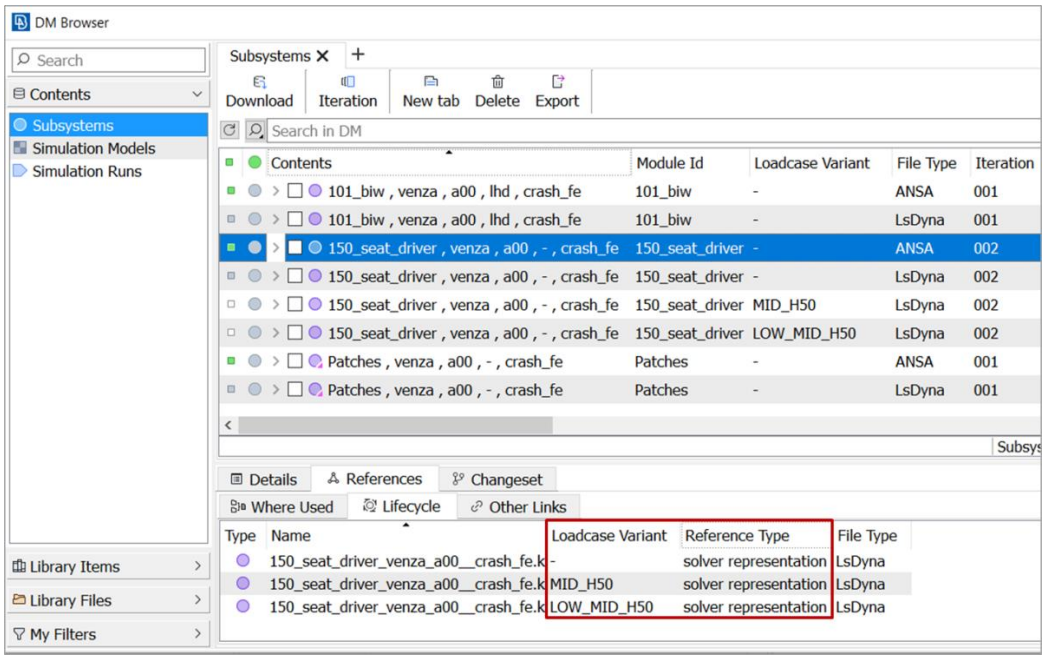
After Save in DM action finishes, the *Alternative* representation of the Simulation Model and Subsystem have been generated according to the "Loadcase Id" property value.



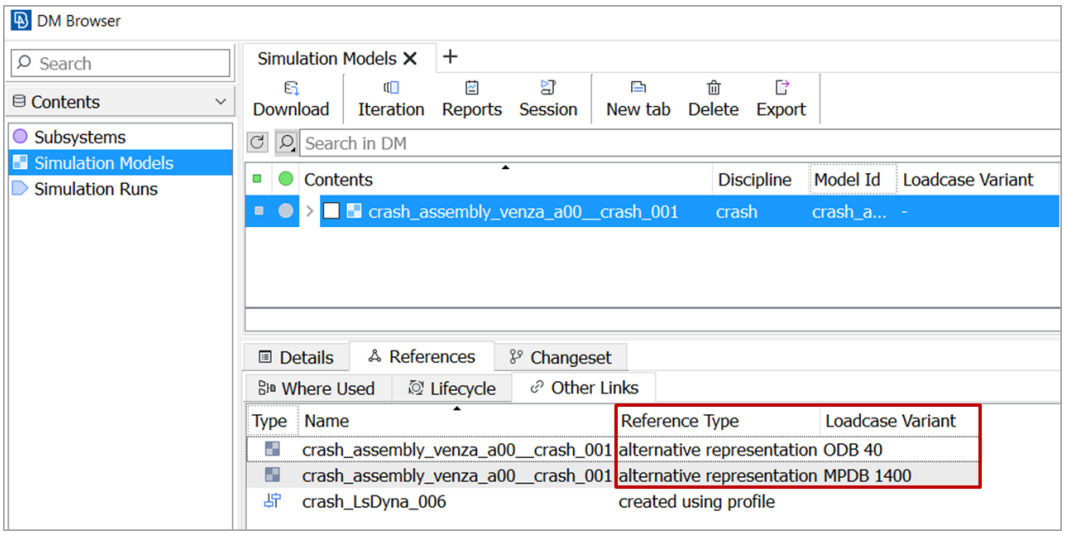
During Build, the ANSA defaults script function ran in the background and assigned the proper Loadcase Variant to the Subsystem and the Simulation Model. As the LsDyna file of the Subsystem was not found in DM with the requested Loadcase Variant, ANSA created it automatically by building the ANSA Subsystem. As the Build Process of the Subsystem includes the user-script "Build" action that positions the seat to the requested position, the seat was properly positioned and the LsDyna file of the Subsystem was automatically saved in DM.

After repeating the process for the various Loadcase Ids, all the required Subsystem representations and the related Simulation Models are created and saved in DM. The Alternative representations of the Subsystem are linked to the *Nominal* one with a solver representation link. By selecting the Nominal Subsystem and switching to the Lifecycle tab at the bottom of the DM Browser, the user is able to see its associated Alternative representations.

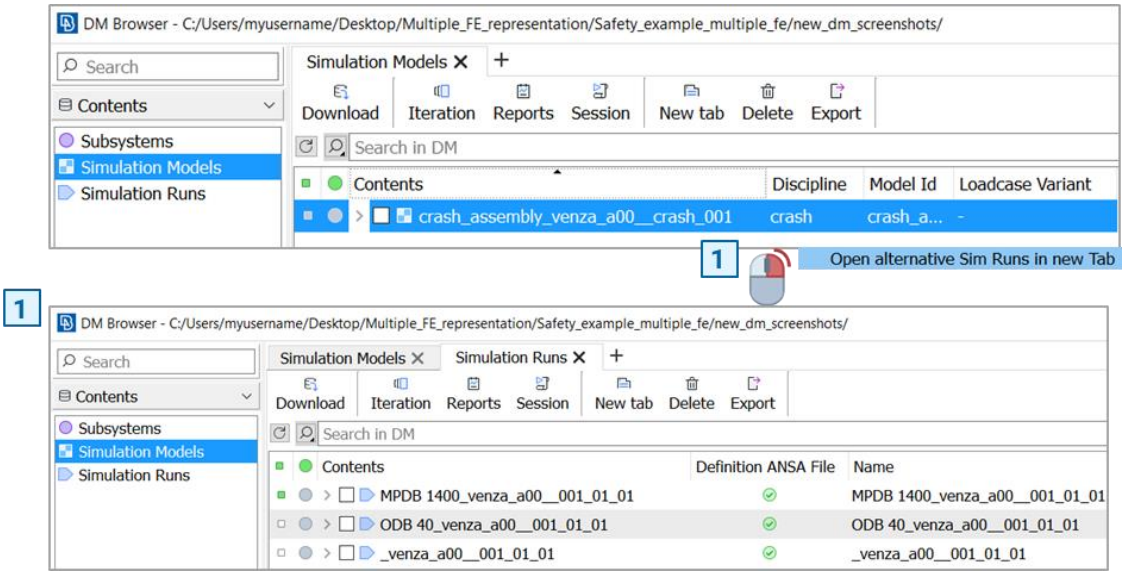
Note! The same process can be applied either on a loaded or an unloaded model.



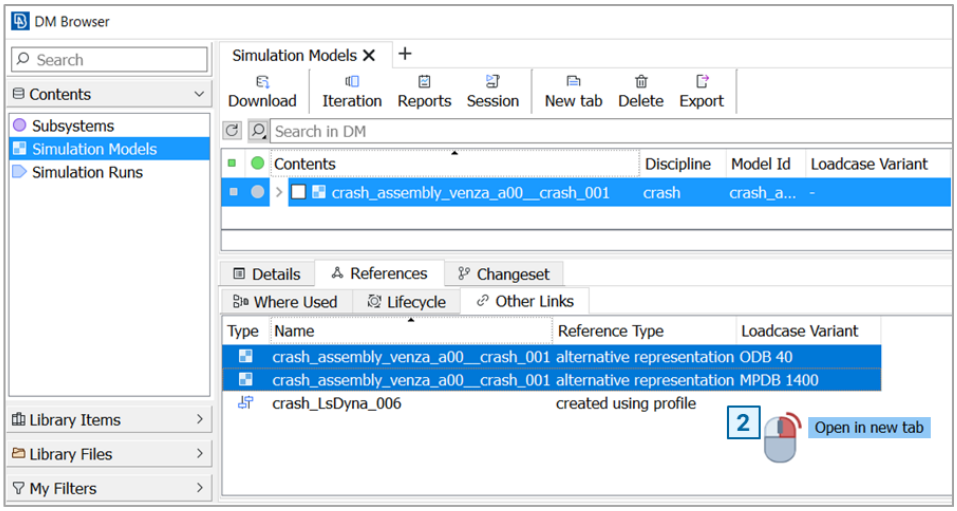
Although a different Simulation Model DM Object is created for each different Loadcase Variant of the Nominal representation, only the Nominal Simulation Model is shown in the top list of the DM Browser. By selecting it, all *Alternative* Representations that are associated with a variation of this model are listed in the ***Other Links*** tab.

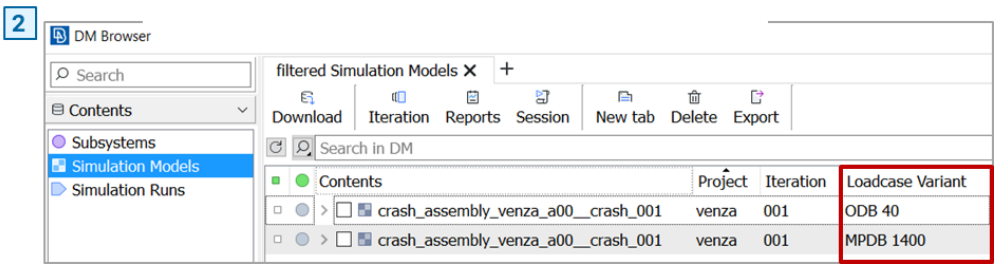


Open alternative Sim Runs in new Tab right-click action can be utilized in order to open the Sim Runs associated to all alternative models in a new tab in the DM Browser.



Although the alternative Simulation Models are not visible in the Simulation Models Tab by default, it is possible to open them in a new tab by selecting them in the References bottom tab and selecting the option ***Open in a new tab***.





11. Getting Started with the Modular Environment

11.1. Configuration

This paragraph describes the main settings that are used for the Configuration of the Modular Environment.

11.1.1. DM Schema

The first step for the configuration of the Modular Environment is the set-up of the DM Schema that controls the characteristics of all modules and compositions.

File-based ANSA DM comes with a built-in DM Schema that can be previewed in the DM Schema Editor. SPDRM also comes with an embedded DM Schema, that can also be previewed in the ANSA DM Schema Editor. The default DM Schema can be adapted to the nomenclature used in each CAE team. For file-based DM, modifications can be made directly in the DM Schema editor. For SPDRM, modifications need to be made manually by editing the configuration file dm_schema_TBM.xml with the guidelines provided in the SPDRM Users' Guide.

Common modifications include:

- Modification of attribute names, e.g. Program instead of Project, Milestone instead of Release, etc.
- Modification of the accepted values of attributes, e.g. accepted values for Module Ids, Projects, Releases, Representations
- Addition or removal of primary attributes for particular DM Item Types, e.g. addition of Body Type, Steering, Roof, Transmission, Market for the identification of Simulation Models
- Addition or modification of the default generation rules that are used to compose the names of the Model Containers based on the values of their primary attributes

Modifications that must be avoided, because they may disrupt the smooth operation of the system:

- Renaming and changing the type of the hard-coded attributes: Module Id, Representation, File Type, Status, Name
- Defining attributes with the reserved names Version, Study Version and assigning them any type other than the default i.e. TEXT and STUDY VERSION respectively
- For Simulation Models, Loadcases, Simulation Runs and any other type that has a primary attribute of type FILE defined last, no other new primary attribute must be added after the file attribute

11.1.2. Configuration on module level

The most crucial part in the configuration of the Modular Environment has to do with the tuning of the settings that affect the creation of the individual modules. These settings control the tools used by model building teams in order to prepare modules that can be directly "consumed" by compound Model Browser Containers.

Marking of Interface Points

An important step is the configuration of the settings that relate to the marking of Interface Points in the Modular Environment. These settings are heavily dependent on the assembly strategy that will be used during the preparation of the composition (i.e. Simulation Model or Loadcase).

There are three questions that must be answered in order to configure properly the settings that relate to the marking of Interface Points, i.e. Assembly Points (A_POINTs) and Loadcase Points (LC_POINTs):

Q1. Must the interface points be named with unique names?

The answer may be different for Assembly and Loadcase Points and for different engineering teams within the same CAE department:

Assembly points are usually too many to have meaningful unique names.

Loadcase Points for durability, fatigue and NVH loadcases usually have unique names.

Loadcase Points for crash, that may mark the position of sensors, may be difficult to get unique names.

Based on the answer, the following settings **merge_A_points**, and **merge_LC_points** must be configured. These settings control the composition procedures that take place during Smart Assembly. They can get 2 alternative values:

merge_A_points = equal name
This value must be used in case the Assembly Points have unique names. It means that during Smart Assembly, if ANSA detects two A_POINTs with the same name, it will merge them into one, no matter their distance.
merge_A_points = equal name and position
This value must be used in case the Assembly Points do not have unique names. It means that during Smart Assembly, ANSA will only merge two A_POINTs with the same name if they are close enough to each other.

merge_LC_points = equal name
This value must be used in case the Loadcase Points have unique names. It means that during Smart Assembly, if ANSA detects two LC_POINTs with the same name, it will merge them into one, no matter their distance.
merge_LC_points = equal name and position
This value must be used in case the Loadcase Points do not have unique names. It means that during Smart Assembly, ANSA will only merge two LC_POINTs with the same name if they are close enough to each other.

Q2. Are the post-processing sessions based on node ids/node names?

The marking of interface nodes with Assembly Points and Loadcase Points can be transferred to META, where these entities are drawn in a characteristic way and all identification functions can work by using these entities as input.

For those cases where there are legacy post-processing session files that already use the ids and names of nodes for identification, it is possible to request each Assembly Point and Loadcase Point to take care of the id, name and comment of the node it finally marks.

The switches below control which attributes of the A_POINT/LC_POINT will be passed to the interface node:

copy_A_POINT_id_to_node
Activating this switch, the number defined in the Node Id field of the A_POINT will be transferred as Id to the Interface Node (and the node will be marked as FROZEN_ID=YES)
copy_A_POINT_name_to_node
Activating this switch, the Name of the A_POINT will be passed to the Name of the Interface Node.
copy_A_POINT_comment_to_node
Activating this switch, the Comment of the A_POINT will be passed to the Comment of the Interface Node.

copy_LC_POINT_id_to_node
Activating this switch, the number defined in the Node Id field of the LC_POINT will be transferred as Id to the Interface Node (and the node will be marked as FROZEN_ID=YES)
copy_LC_POINT_name_to_node
Activating this switch, the Name of the LC_POINT will be passed to the Name of the Interface Node.
copy_LC_POINT_comment_to_node
Activating this switch, the Comment of the LC_POINT will be passed to the Comment of the Interface Node.

Q3. Will the intermodular assembly method require modification of the A_POINT representation element (Type B connections of paragraph 6.1.2)?

Assembly Points that are auto-generated by Bolt Connections are associated with an interface element, usually an RBE2 or nodal rigid body or a rigid patch. Since the A_POINTs belong to the modules, saving the module will take these elements along. In case the connector that will use this A_POINT during intermodular assembly will need to modify its element, for example in case the connecting element must be a zero-length element, where the coordinates of the interface nodes from the two sides will have to be matched, or in case the connecting element must generate a rigid element in Abaqus, Ls-Dyna, Pam-Crash or Radioss, where a rigid dependency error will be generated and will have to be resolved, this won't be possible unless a new version of the module is generated. Since this wouldn't be practically applicable, it is possible to mark the A_POINTs in a way that will enable such modifications to take place without affecting the file of the module, by suppressing the A_POINT elements from the keyword file of the module and saving them in a separate file that is managed automatically during Smart Assembly.

So, depending on the assembly method, the default behavior of the A_POINTs and LC_POINTs can be controlled through the setting **output_with_subsystem**.

output_with_subsystem = true (default)
When this setting is activated, any elements that have been generated by or belong to the A_POINT/LC_POINT will be written out when the module will be saved as a solver keyword file.
output_with_subsystem = false
When this setting is deactivated, any elements that have been generated by or belong to the A_POINT/LC_POINT will not be written out when the module will be saved as a solver keyword file.

Numbering

The Modular Environment utilizes the capabilities of the Renumber Tool in order to control the ids of entities on modular level. Thus, it is possible to define an *.ansa_rules* file in order to define a default numbering range for each module, based on its name. The defined numbering rules are applied through the Build Process, from the action *Source Numbering Rules*. However, it is also possible to set a proper id to entities right at the moment they are created. The settings that control the numbering of entities are provided in the table below:

default_numbering_rules_file = <file path of .ansa_rules file>
Through this setting it is possible to specify an *.ansa_rules* file that contains special rules for the numbering of includes used by Model Browser Containers. In this file, using a regular expression based on the include's name guarantees that the moment a new module is created and some entities are added to it, a numbering rule is automatically created. The file path can be an absolute path, a path inside the current DM (DM:/) or a path defined with environmental variables. A sample of such file is given below. The expressions have been highlighted in bold:
INCLUDE NAME: **"100_BIW*"** SPECIAL RULE: "General Rule for all groups of entities" <ALL GROUPS> = 10000001 : 19999999 : 1, PRESERVE INCLUDE NAME: **"201_DOOR_FL*"** SPECIAL RULE: "General Rule for all groups of entities" <ALL GROUPS> = 20100000 : 20199999 : 1, PRESERVE INCLUDE NAME: **"202_DOOR_FR*"** SPECIAL RULE: "General Rule for all groups of entities" <ALL GROUPS> = 20200000 : 20299999 : 1, PRESERVE INCLUDE NAME: **"203_DOOR_RL*"** SPECIAL RULE: "General Rule for all groups of entities" <ALL GROUPS> = 20300000 : 20399999 : 1, PRESERVE INCLUDE NAME: **"204_DOOR_RR*"** SPECIAL RULE: "General Rule for all groups of entities" <ALL GROUPS> = 20400000 : 20499999 : 1, PRESERVE
create_new_ids = from_numbering_rules, copied_entity_id = from_numbering_rules
These values must be used in order to request all new created entities to get their ids directly from the numbering rules of the Model Browser Containers they belong to. This setting is very important because it is required in order to control the ids of entities that may be created automatically by ANSA during output.

11.1.3. Configuration on composition level

There are a couple of settings that affect the operation of the program when working with compound containers like Subsystem Groups, Simulation Models, Loadcases and Simulation Runs.

get_always_current_part = true
When this setting is activated, any new created entity will be automatically assigned to the Model Browser Container that is marked as *current*.This way, it is possible to control the destination container of new created entities when working in an ANSA session that contains several Subsystems or Library Items.

auto_target_id_range_method
This setting controls what will happen the moment a module is added to a compound Model Browser Container. The following values are supported: **Always ask**: The Target ID Ranges tool will open, to prompt for the definition of the target id ranges to be used in the adapter **Keep Source**: The target id ranges will be automatically defined based on the source **Skip** / **Define later**: The target id ranges will be left blank. The user will be prompted to fill them in during Build.

11.1.4. Save settings

In the Modular Environment it is possible to pre-configure the save settings used during “Save in DM” of the Model Browser Containers, so that the default values suggested to the end-user are those that facilitate the Simulation Run composition, as this was described by the team leader. As described in paragraph 3.7, the “Silent Save” option can also be used in order to silence any pop-up windows during the saving process, and let the program work with a predefined save scenario. The main settings that relate to modular save are described in section 3.7.

Some additional settings that relate to the Save process:

additional_model_set_up_types_for_Subsystem_save
With this setting it is possible to save along with a Subsystem entities that do not belong to it, because they cannot be added to an ANSA Part or to an Include entity, e.g. Connection Templates, Morph Boxes, Lock Views, etc. In order to activate this behavior, set a comma-separated string of ANSA types that need to be saved along with the subsystem (e.g. CONNECTION_TEMPLATE, MORPHBOX)

allow_reference_of_inner_subsystems_of_groups_instead_of_group_subsystems
Activating this setting, the main keyword file of the Simulation Run will contain include keywords that reference the Subsystems contained in the Subsystem Groups of the Simulation Model and/or Loadcase and not the Subsystem Groups themselves.

make_comment_mandatory_for_save_iteration_of_subsystem ***make_comment_mandatory_for_save_iteration_of_library_item*** ***make_comment_mandatory_for_save_iteration_of_simulation_model*** ***make_comment_mandatory_for_save_iteration_of_loadcase*** ***make_comment_mandatory_for_save_iteration_of_simulation_run***
Activating this setting forces the Save procedure to request the definition of a comment when a new iteration is about to be created

dm_status_values_allowing_overwrite
This setting accepts as a value a string of comma-separated status values that must be considered as indicators that the compound Model Browser Container is in debugging phase. dm_status_values_allowing_overwrite = Draft For example, if the Simulation Model in DM has status Draft, then its contents can be overwritten. Otherwise, not.

BETA CAE Systems International AG
D4 Business Village Luzern, Platz 4
CH-6039 Root D4, Switzerland
T +41 41 545 3650, F +41 41 545 3651
ansa@beta-cae.com
www.beta-cae.com

physics on screen