

Description of the TKC Toolkit Creator

An MSC.ADAMS based Environment for Rapid Design and
Management of Custom Simulation Toolkits

Chris Verheul

2/25/2014

This manual describes the basic assumptions made in TKC, the additional GUI elements and advised working method of TKC for defining advanced model component toolkits for MSC.ADAMS. Users following the basic rules described and the TKC features added to the standard MSC.ADAMS/View graphical environment will drastically increase their efficiency in defining, maintaining and improving custom toolboxes of project specific simulation components.

Contents

Installation of TKC	3
Installation and first ADAMS TKC session	3
The TKC Standard Toolbar	5
About TKC and TKC-based Toolkits	6
More on TKC UDEs	6
C_UDE or Component-UDE	6
D_UDE or Data-UDE	7
Sub Components	7
TKC features and properties	8
Properties and aspects of a TKC based Toolkit	8
Example Toolkit Component: MBS (Multi-Body Samples)	9
TKC File and Directory Structure.....	11
User Project level file and directory structure	11
Site level file and directory structure	11
Shared level file and directory structure	12
Toolkit level file and directory structure	12
TKC System level file and directory structure	13
The TKC Graphical User Interface (GUI)	14
The Toolkit Dialog.....	14
The generic Component or C_UDE dialog.....	15
The generic Component Data of D_UDE dialog.....	16
Typical aspects to consider using UDEs and macros	17
Placeholders used in this documentation	18
Creating and Using a new Toolkit Component.....	19
Introduction.....	19
Description of the Component.....	19
Building the non-parameterized component.....	20
Parameterization of the component structure	22
Register the Component as a TKC C_UDE.....	27
Registration of the Component and its Interface	27
Definition of the Component Creation Macros	28

Installation of TKC

You have received a zipped file with a working directory of a TKC project. The files to define all TKC functionality are included and are defined to set up a so-called *local* installation. This means that all names and directories are defined relative to the current working directory. Users can move and rename directories after the installation as long as correct changes are made in the respective *.cfg files. For more understanding of the TKC system files and directories we refer to the files section in this manual.

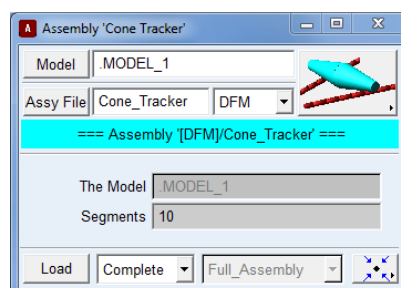
Installation and first ADAMS TKC session

1. Unzip the Zip file into a working directory (say *C:\My_Work*).
2. Check the files and directory tree in the working directory (read the files section).
3. Start ADAMS with the current directory as the working directory (or create a shortcut on your desktop in Windows so you start ADAMS with the desired working directory as start-up directory).
4. The ADAMS system file *aview.cmd* will define an additional button on the main menu enabling you to load the TKC system. You can rename *aview.cmd* (i.e. to *aview_.cmd*). In this case the button will only appear after manually loading this *.cmd file.
5. In the first session, ADAMS defines all TKC macros and dialogs and creates a partial binary. This may take some minutes (depends on machine speed). In following runs, TKC loads from the bin file in a few seconds. The partial bin file can be deleted either from the *TKC* menu or manually.
6. The graphical user interface (GUI) of TKC toolkits use the ADAMS *Standard Toolbar* (from now on this *TKC Standard Toolbar* will be named *TST*). Right clicking on the MSC logo on the right hand side will open a toolkit specific menu.

Loading your first assembly model (A Cone on two discrete beams) is done using the following sequence of clicks:



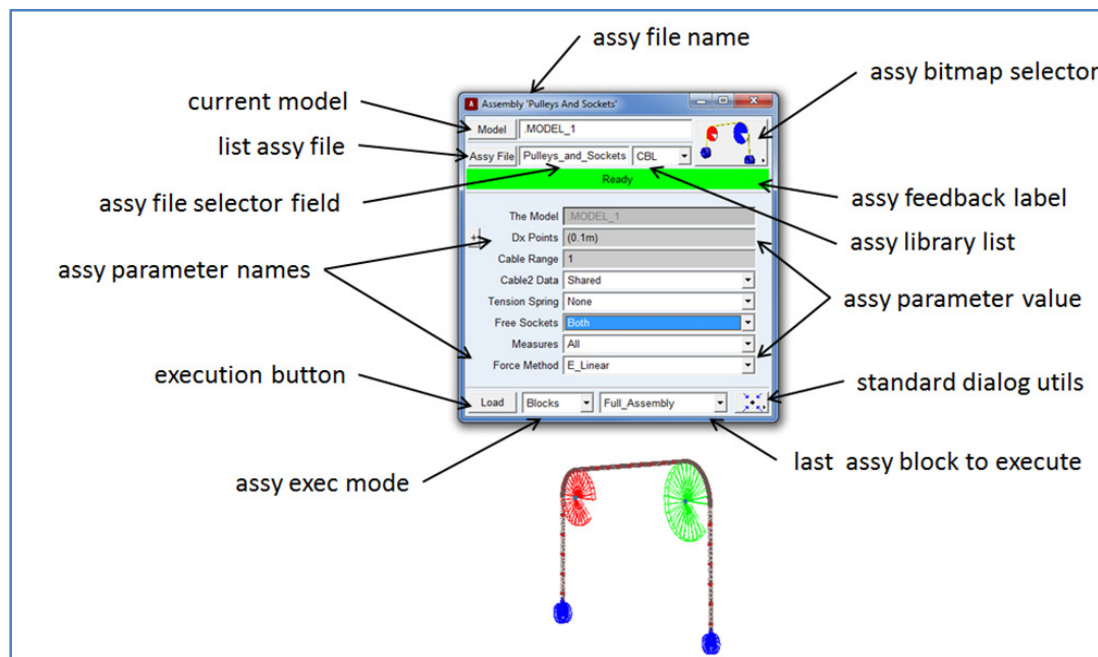
Click on the Assembly Loader button in the TST: and Adjust the Assembly Loader Dialog input fields so it is identical to the picture below. The different elements in the dialog will be explained later in this manual.



Basically it means that you want to load the assembly *Cone_Tracker* from the Shared *DFM* Toolkit Assembly library into the existing *Model_1*. Before loading this model assembly, the number of segments in the two discrete flexible tracks can be set using the *Segments* parameter of the assembly.

Click on *Load* in the Assembly dialog to load the assembly into the current model. The model behaves like it would in plain ADAMS. You can start changing it or run a simulation directly as you would in a normal ADAMS session.

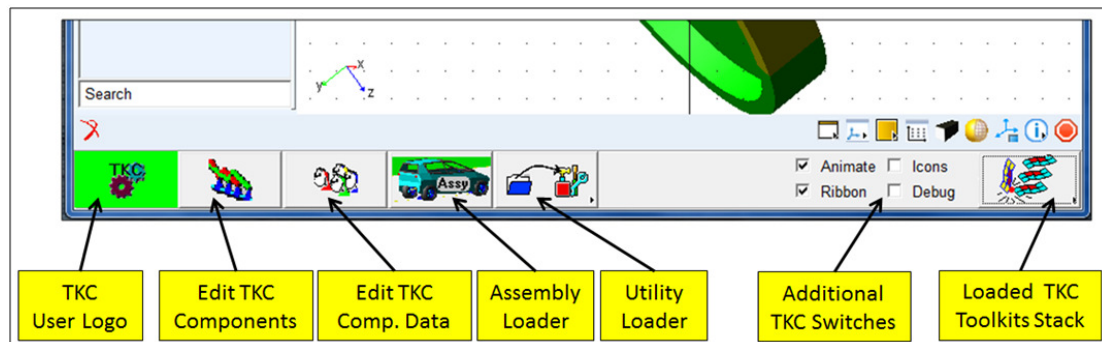
The assembly dialog is quite loaded with functionality. Much coding has been invested in this dialog, also because it is completely generic and is one of the 4 main dialogs used throughout TKC. The image below illustrates the dialog for a cable based assembly file.



Overview of the functionality of the TKC Assembly Loader Dialog

The TKC Standard Toolbar

The TKC Standard Toolbar (TST) contains a number of often used buttons to manage objects, data, utilities and assemblies:



From left to right the TST contains:

- The *TKC User Logo*, an info string is shown on *mouse over*,
- The *C_UDE List button* shows the list of existing TKC component objects in the current model. After selection of a C_UDE, the C_UDE modify dialog will be displayed.
- The *D_UDE List button* shows the list of existing TKC component data objects in the current model. After selection of a D_UDE, the D_UDE modify dialog will be displayed.
- The *Assembly Loader button* opens up a dialog to open a special group of TKC assembly macros. Assembly macros (with file extension *mac_assy*) contain lines creating D_UEs and C_UEs to create a complete model assembly.
- The *Utility Loader button*. Utilities in TKC are macros from one of the TKC libraries. Typically a TKC utility is an enhanced ADAMS macro (with file extension *.mac) which is loaded dynamically from a macro file browser. Standard Utilities are a group of utilities that users can put together as their personal favourite. A stack of icons is presented for more intuitive loading of these utilities.
- The different Toolkit specific dialogs for all loaded TKC toolkits can be displayed from the button stack stored in the *Toolkit Dialogs button*. A Toolkit specific Dialog is a generic dialog containing buttons to display dialogs for creating toolkit specific C_UEs, and/or D_UEs or executing toolkit specific Utilities.

About TKC and TKC-based Toolkits

The TKC toolbox is a set of macros and dialogs defined in the ADAMS/View command language and dedicated for models in ADAMS. The purpose of the TKC toolbox is to manage, maintain and document a user toolkit. The TKC toolbox creates model objects called C_UDEs (for components) and D_UDEs (for data objects). The definition of C_UDEs and D_UDEs resembles the so-called ADAMS UDEs (User Defined Entities). A UDE can be considered as a component in a model, possibly containing a hierarchy of other (sub-component) objects. Some advantages of using macro defined UDEs in models are:

- Allowing for structured modeling,
- Automating the modeling work so more complex models can be created,
- Capturing the proper dynamic modeling of component dynamics in user macros for certain model building blocks often used in certain companies or projects (i.e. knowledge capture).

In ADAMS, users can define new UDE types to define a more readable model structure. In TKC this functionality is further automated leaving the user with simple interactive and intuitive modeling tasks, even when creating and modifying UDEs and user dialogs in a toolkit. Thus, several toolbox development and debugging tasks are supported in a TKC-based toolkit.

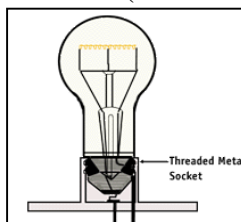
More on TKC UDEs

The main purpose of TKC is to offer an environment in ADAMS that supports the advantages of using macros for model components and does not restrict users in working with macro based models. One of the main reasons for not using the official ADAMS UDE objects in TKC is because they are stored as entities in the ADAMS binary. For this reason, models containing UDEs can only be loaded into an ADAMS session that already has the correct UDE definition loaded in the main database (aview.bin). As this was considered to be restricting, the UDE concept was aborted during the development of TKC. Instead, all C_UDE and D_UDE objects are defined as standard ADAMS design variables with are flagged by the presence of string variable children with reserved names (*C_UDE_Type* and *C_UDE_SubType*) to define the component type and subtype (i.e. *.Model_1.Strut_YPos.C_UDE_Type* and *.Model_1.Strut_YNeg.C_UDE_SubType*).

Two main UDE types are defined in TKC: C_UDEs and D_UDEs. The main use of C_UDEs and D_UDEs will be discussed shortly:

C_UDE or Component-UDE

This type of UDE is used in TKC to store (or contain) equations and objects typically for model Components. The difference between C_UDEs and *normal* ADAMS UDEs is explained by a light bulb example. In normal UDEs the model components (parts, forces etc.) used to model it are children of the UDE in the ADAMS database. In a *light bulb* C_UDE, only the definition and interface of the



difference between C_UDEs and *normal* ADAMS UDEs is explained by a light bulb example. In normal UDEs the model components (parts, forces etc.) used to model it are children of the UDE in the ADAMS database. In a *light bulb* C_UDE, only the definition and interface of the

socket are defined as children of Design Variable *MyBulb*. The typical use of variable *MyBulb* is indeed to be the interface for the light bulb C_UDE, so macros and utilities can communicate to it and connect it to other C_UDEs (i.e. the fitting for positioning and input of electric power).

For a light bulb, there might be an input parameter named *Reference* of type Marker in the definition of the light bulb C_UDE. This input parameter *Reference* introduces the name of an existing marker to light bulb UDE instances. By passing the marker object *Reference* to the internal structure of *MyBulb*, equations can be defined to create the fitting and the glass of the bulb. All other components of the light bulb (including the light emitting parts, i.e. the parts that do the work), are defined in a C_UDE creation macro (with extension *mac_cre*) using a *Comp__Base* naming method. In this method, *Comp* represents the name of the C_UDE (in this case *MyBulb*) and *Base* is the name of the light bulb component (i.e. the glass part). The two underscore characters cannot be omitted and the complete name thus links the (glass part) object to the correct light bulb C_UDE. Thus, if an ADAMS part would be created for the glowing spiral of the bulb, its name could be defined as *MyBulb__PartSpiral*.

D_UDE or Data-UDE

This type of UDE is used in TKC to define model parameters used by C_UDEs. For the light bulb C_UDE, typical parameters are the geometry of the light emitting elements, (i.e. the radius of the glass and possibly the resistance and other properties of the wires). Thus all parameters specific to a C_UDE are stored in one Model object with a unique D_UDE Type.

For a round shaped Light_Bulb of toolkit EFC (Electric Furniture Components) the definition code of the Data object is defined as *EFC.Light_Bulb.Round.Data*.

The Data Type and Data Subtype definition of a light bulb data object are stored in strings variable children of the light bulb data object:

- *.Model_1.Data_My_Bulbs.D_UDE_Type* String = "*EFC.Light_Bulb*" for the Data Type definition
- *.Model_1.Data_My_Bulbs.D_UDE_SubType* String = "Round" for the Data Subtype definition

As was already described in the text, the *D_UDE_Type* and *D_UDE_SubType* string are used as flags to denote that:

- *.Model_1.Data_MyBulbs* is a D_UDE object with data for one or more C_UDEs,
- The parameters are typically defined to be used by one or more C_UDE objects of type *EFC.Light_Bulb.Round*. Different C_UDE model instances, all using the same D_UDE object will therefore have identical parameters such as radius, color and other parameter settings. Moreover, changing the data in *Data_MyBulbs* will automatically change the characteristics of all C_UDEs using this D_UDE as they are all parametrically attached to it.

Sub Components

The use of sub-components is a recent extension of the TKC functionality. Further explanation is available in animations of typical sub-component definition sessions.

TKC features and properties

The functionality and added value of TKC has many usage oriented items. The following list shows some of the starting points for developing TKC and thus increasing the user experience of ADAMS Multi-body simulation users.

- Minimal actions are required to define, register and update ADAMS toolkits.
- In TKC, generic dialogs are used as GUI for managing of C_UDE and D_UDE instances. In effect, there is no need to create and maintain any user dialogs for new defined Component types in a toolbox. In case users need special dialogs, i.e. with cross linking of fields and special objects such as radio buttons, they can be created and maintained in the same TKC toolkit directory. In general however this is not advised as the work in designing and maintaining user dialogs can be quite time intensive.
- All generic dialogs used to manage C_UDE and D_UDE instances are linked to the on-line help files defined in the toolkit.
- TKC allows for deleting, copying and editing all toolkit components. Thus, new UDE entity types can be defined and registered by making a copy of existing UDEs and making the desired modifications. All *toolkit edit actions* can be done on-line in an interactive ADAMS modeling session. As all toolkit data is stored in ASCII files, it can also be done manually. Toolkit functionality is available once the toolkit name is registered in the tree of TKC configuration files.
- Updating of model modifications in a C_UDE is supported interactively. The method is based on automated writing of C_UDE model lines to the C_UDE macro file using the ADAMS macro syntax. This functionality prevents a large part of the hand-based work (typically starting from copying and pasting from the model *.cmd file) that is classically required when making a macro in ADAMS. Another advantage is that the macros are written in a strict syntax allowing for extra checks when loading and defining the UDEs as model components.
- A large part of the on-line documentation for a toolkit is created by extraction and re-arranging of ASCII data from the toolkit definition. For this purpose, TKC has a tool called *UDE_Manual* and dialogs *UDE_Visual* and *UDE_Manual*.

Properties and aspects of a TKC based Toolkit

Definition and storage of a TKC based toolkit is based on an intuitive set of rules. In the following list of toolkit properties we assume a toolkit named *MyKit*.

- All toolkit files are in ASCII format in a single toolkit directory. This toolkit directory typically resides in the *TKC_Site* directory and is named *MyKit*.
- The main functionality and list of C_UDEs and D_UDEs in a toolkit and their respective interfaces is stored in the toolkit configuration file *_MyKit.cfg*.
- If the name of a certain macro or tool is not included in this file, it is not registered. Thus, UDE or utility files can exist in the toolkit without being visible to the user. Only after being registered in *_MyKit.cfg*, they can be activated from the TKC dialogs.
- A strict naming convention is applied for all files and objects in a toolkit. In the example below this is illustrated.

- The type name of a C_UDE object is *[MyKit].[MyType]{.[MySubType]}*. The complete type name of a D_UDE is *[MyKit].[MyType]{.[MySubType]}.Data*. The {} indicates that UDE objects can also be defined with a blank Subtype.
- The name of the macro file defining this component object is:
`[MyKit]_[MyType]_{_[MySubType]}.mac_cre` in the root of directory *MyKit*,
- The name of the macro file defining the data object is:
`[MyKit]_[MyType]_{_[MySubType]}.mac_dat` in the same directory.
- In the creation process of a C_UDE in a model, the typical sequence is:
 1. The user selects which sub-type of C_UDE he wants to model (i.e. *Rigid* or *Flex*) and creates the appropriate D_UDE object (or refers to an already existing model D_UDE object).
 2. The C_UDE will be created, referring to the D_UDE. In all C_UDE objects, a parameter named *MyUde.Data_Name* holds the name of the D_UDE object used.
- No limit exists for the number of C_UDE instances referring to a single D_UDE instance. Thus, a minimal amount of design variables is defined in a model. Also, all C_UDEs referring to a D_UDE will be updated automatically when design variables in the D_UDE are changed.

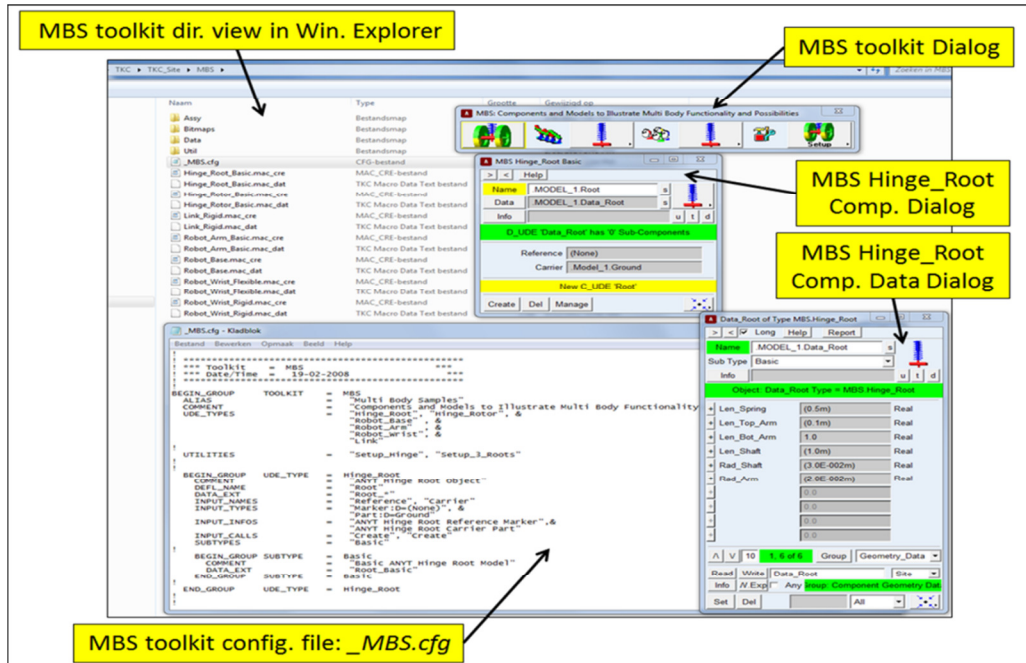
Example Toolkit Component: MBS (Multi-Body Samples)

Suppose a toolkit *MBS* is defined to store and use simple multi body components. One of the components is a robot arm fixed to a support frame. Of this robot arm model, a rigid version exists that is modeled as three un-deformable parts linked with ideal frictionless joints. This arm model is parameterized with a limited number of design variables and can be used for simple exploratory simulations. Also, a more complex version (sub-type *flex*) exists of the robot arm. In this version, the bodies are defined as flexible using a discrete flexible approach. The flexible structures are linked together at the arm hinge locations using non-linear spring-damper elements with friction. This model was actually made after learning the basics from the rigid arm and was defined by adding complexity to the rigid arm model components. Finally, the flexible arm model is stored as a separate C_UDE entity so it can be used in parallel with the simple rigid version.

This results in the following toolkit components:

- UDE entity *MBS.Robot_Arm* with subtypes *Rigid* and *Flex*. Only one C_UDE type (*MBS.Robot_Arm*) is defined for the two sub-types (to check: use *info* on the UDE object). The reason for this is that the UDE only contains the interface variables (i.e. *Carrier* to define the frame to which to connect the arm to). The users can verify which arm type has been modeled from the UDE variable *.Model_1.MyArm.C_UDE_Subtype*. The value of this string type design variable will either be *Rigid* or *Flex*.
- Using a strict naming convention, the arm model elements (parts, forces etc.) are defined as entities belonging to the C_UDE *MBS.Robot_Arm*. This means they are all named
 - *.Model_1.MyArm__[XX]* (for parts, forces, constraints etc.) or
 - *.Model_1.PartName.MyArm__[YY]* (for part children such as markers, points and geometries)

- The design variables to define geometry and force components (stiffness, damping etc.) for *MBS.Robot_Arm* C_UDEs are stored in D_UDE arrays of type *MBS.Robot_Arm.Rigid.Data* or *MBS.Robot_Arm.Flex.Data*. Due to the different modeling methods used, different parameters are required for the two C_UDE subtypes (i.e. a *Flex* Arm requires other variables than a *Rigid* Arm).



Directory structure, files and dialogs of a typical TKC toolkit

TKC File and Directory Structure

Files and directories for storage of TKC components are defined at five levels:

1. **User Project level** : where specific data is stored for individual users or projects (i.e. which toolkits to link, what look-and-feel to apply, a directory with user model data, a set of project specific utilities),
2. **Site level**: where toolkits and data directories are defined applicable for all users in a so-called Site. Typically this is used to share toolkits between colleagues,
3. **Shared level**: in this level all commonly distributed TKC toolkits and data are stored. The GUM (General Utility Macros) is stored in this level as well as some utilities and data files (typically used for samples),
4. **Toolkit level**: all toolkit functionality is stored in a toolkit directory, the name of this directory is identical to the name of the toolkit,
5. **Code level**: to define all code used in the TKC toolbox.

Files and sub-directories in each TKC level will be briefly discussed.

User Project level file and directory structure

The following files and directories are typically present in a TKC/ADAMS working directory. (Note: most ASCII files mentioned can be text edited and/or exported from ADAMS):

- **_TKC_User_Data.cfg** (file) All TKC system information is stored in a tree of configuration files. This tree starts with the file *_TKC_User_Data.cfg*. This file either exists for each ADAMS user in the user Home directory (i.e. *Documents and settings/user*) or for each project separately (in the user project work directory). Besides user specific data, this file defines the TKC system root directory. On default, this directory contains:
 - TKC_Shared: the directory with Shared TKC toolkits and other data,
 - TKC_Site: the directory with Site unique ADAMS toolkits (i.e. Company proprietary information),
 - TKC_Code: the directory with all TKC macro and command files
- **aview.cmd** (file) a standard file name used by ADAMS to load certain data in an ADAMS session. A standard version of this command file is included in the TKC package to load the user data and define a button to load the remainder of the TKC functionality. This file can as well be stored in the user Home directory.
- **_Data** (dir) Contains model data files to be read by D_UDEs. Data for each D_UDE can be written and read to files with a fixed extension. User libraries can be defined as sub-directories of **_Data**.
- **_Assy** (dir) Contains assembly files to create complete macro structures in ADAMS models. Assembly files contain lines defining D_UDEs and C_UDEs (possibly to create a complete model) in the ADAMS/View command language. Users are advised to work from assembly files for creating new models. The *Make Assembly* utility can create new assembly files from existing models.
- **_Util** (dir) Contains project specific utility macro files for a range of utilities such as changing project specific model parameters or commands to automatically perform tasks (modelling, simulation and post processing).

Site level file and directory structure

The Site level in TKC is used for all macro and data files that are typically used by ADAMS users in a company. It contains a selection of all TKC Site toolkits. For

confidentiality reasons, users typically receive a minimised version of the Site directory only containing directories with user specific toolkits.

- **TKC_Site** (dir) The directory with *Site* TKC toolkits. User toolkits are stored in separate sub-directories of TKC_Site. Two *.cfg files are stored in **TKC_Site**.
- **_Site_Data.cfg** (file) A configuration file to define all Site specific TKC settings. The cfg files are readable and are briefly documented using ADAMS comments
- **MyKit** (dir) all TKC Site toolkits are stored in sub-directories of the TKC_Site directory.

Shared level file and directory structure

The shared level in TKC is used for all data supplied with the TKC package. This includes:

- **_Assy**: (dir) contains shared sample assembly files,
- **_Data**: (dir) contains a tree structure of data files for a range of C_UDE types,
- **_Util**: (dir) contains shared utility macros,
- **GUM**: (dir) A toolkit with a wide variety of **General Utility Macros** and components. Available components are for defining component-to-component contact as well as a more elaborate spring component.
- **CAD**: (dir) A toolkit with specific utilities and model objects to support working with **CAD** package based geometry data.
- **CAMS**: (dir) A toolkit defining typical (force based) machinery components such as pin-in-slots, linear guides with friction and **CAMS** for different purposes,
- **DFM**: (dir) A toolkit defining flexible objects defined using a **Discrete Flexibility Method**. Available components are straight and curved struts and plates.
- **EMO**: (dir) A toolkit for **Executable Model Objects**. Objects in this toolkit have the purpose to define run- and post processing type data and actions in a model. Using EMO objects, users can include an automated simulation and reporting processing structure in a model.

Toolkit level file and directory structure

Depending on toolkit functionality defined, the following files can be found in the directory of a TKC toolkit:

- **_MyKit.cfg** (file) a toolkit configuration file defining all toolkit components.
- **_MyKit.bin_[Vers]** (file) the toolkit partial binary for storage and rapid retrieval of all toolkit definitions, utilities and dialogs. As ADAMS binaries are not always version independent, the extension is completed with a lower case ADAMS version string (i.e. *2013_1* or *2014*) denoting in which ADAMS version the binary is defined.
- **Bitmaps** (dir) TKC supports automatic creation of bitmaps on buttons and other applicable GUI elements. Each C_UDE and D_UDE creation button and toolkit tool button will be illustrated with the appropriate bitmap in case a bitmap file exists. The format used for bitmap files in the ADAMS GUI is the *.xpm ASCII format, which can be created with programs such as Icon_XP. The base names of the bitmap files is identical to the base names of the *.mac_cre macro files.
- **Help** (dir) The directory where TKC expects on-line help files to be. In the current TKC version, file *MyKit.pdf* or file *Mykit.txt* will be loaded if available.
- ***.mac_cre** (files) Macro files for definition of C_UDE types. The files are defined in a strict syntax to allow for automated checks on macro parameters. UDE definition macros are seen as *products* of the TKC toolbox. Thus, documentation of the macro contents has a high priority. The *.mac_cre and *.mac_dat files can be

defined automatically from a model. The readability of the macro is even maintained while making changes to C_UDEs and D_UDEs. Thus, the macro files are quite useful for (automated) documentation purposes.

- ***.mac_dat** (files) Macro files for definition of D_UDE types. D_UDE instances are data arrays in a model. The children of a D_UDE represent design variables (i.e. mass, stiffness, positions) to be used by C_UDEs. The following naming convention is used for the macro files:
 - *[MyKit]_[MyType]_[MySubType].mac_cre* to define the C_UDE,
 - *[MyKit]_[MyType]_[MySubType].mac_dat* to define the D_UDE.
- ***.mac_del** (files) The deletion of a C_UDE instance may result in errors when objects are deleted that are referred to in other model objects. To prevent this, an optional *.mac_del macro file can be defined to disconnect the C_UDE instance from other model objects prior to its deletion.
- **Dialogs:** (dir) Utilities in a toolkit can use dedicated dialogs. The required *.cmd file to define the dialog is stored in the dialogs dir. (i.e. the dialog for the utility *MDT.MyUtil* is defined in the file *MDT_MyUtil.cmd*).
- **Util** (dir) Toolkit utilities are macros to perform certain tasks (i.e. setting initial velocities to a selected range of model entities).
- **SubComp** (dir) A directory containing *.mac_* files for sub-components defined in this toolkit. Sub components are a further level of model building blocks.
- **Data** (dir)
- **Assy** (dir)

TKC System level file and directory structure

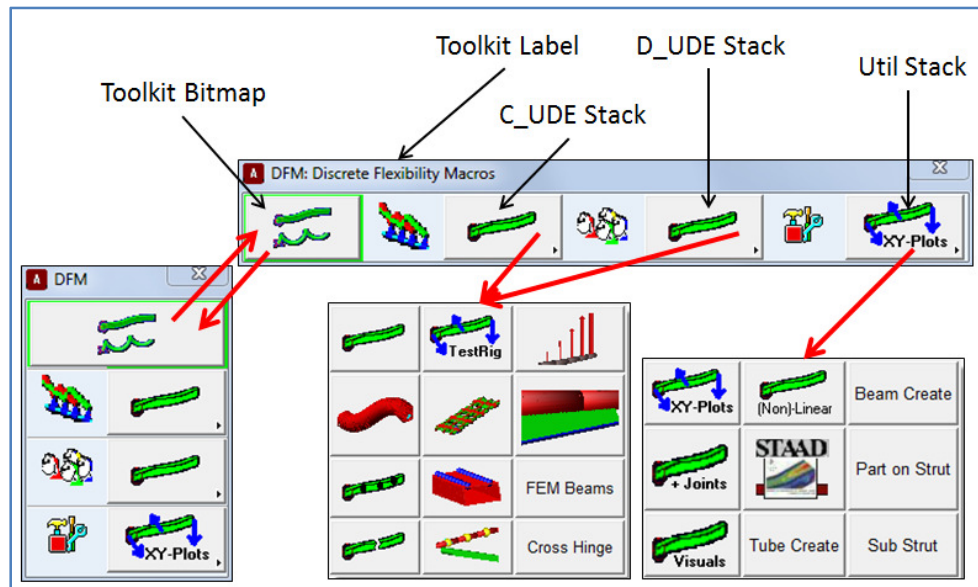
A selection of the files and directories in the TKC toolbox is listed. Users typically will not see or edit these files, unless for code reference or modifications to the TKC functionality. Note: no guarantee of proper functionality can be given after users have made modifications to these system files without consent of the TKC developers.

- **TKC_Code** The library with all TKC macros and dialogs,
- **TKC_Code/TKC** The core TKC macro functionality.
- **TKC_Code/TKC/_TKC.Bin_[Vers]** A partial binary file with all TKC macros, dialogs and other functionality. The file can be created, deleted and updated in any TKC session. You can delete the TKC partial binary by: TKC, Lib Management, and Del TKC Binary. The binary file can also be deleted manually. After recreating the binary file, all TKC core functionality will be updated from all ASCII based macro files.

The TKC Graphical User Interface (GUI)

The Toolkit Dialog

All Functionality of a TKC toolkit is stored in a generated toolkit dialog box.



TKC Shared Discrete Flexibility Toolkit dialog Box

The picture above is a summary of the functionality stored in the DFM shared TKC toolkit. The red arrows denote the response of the dialogs after a left-click on the toolkit bitmap (horizontal-vertical swap) and after right-clicking the button stacks for components, component data and utilities. Each bitmap in the button stacks shown activates creation of a component (C_UDE Stack), creation of data for a component (D_UDE Stack) or opening the dialog for a utility macro (Util Stack).

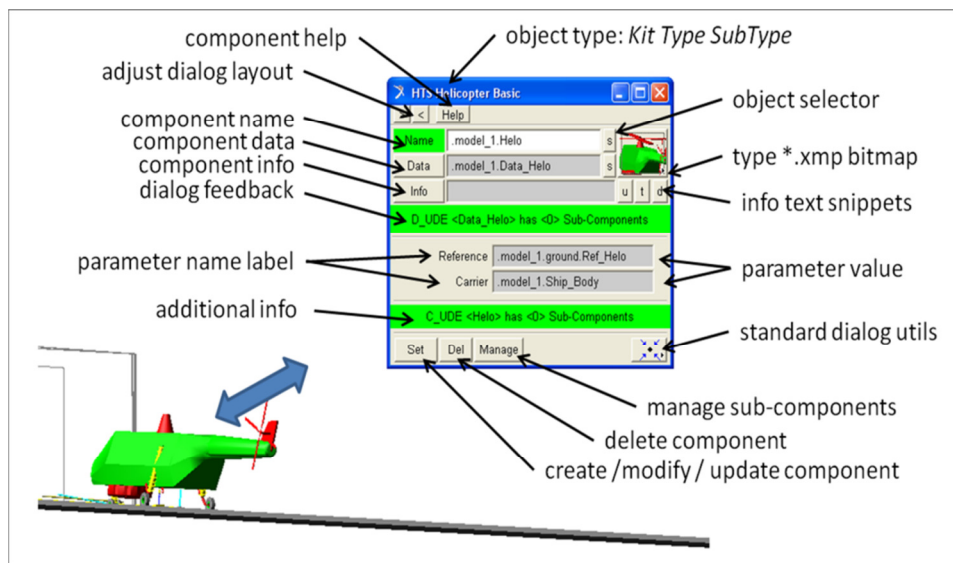
ADAMS button stacks have the following features:

- An ADAMS button stack GUI element can be recognised from the small black triangle at the right hand side bottom.
- When the right mouse button is used on top of a button stack, a matrix of buttons is shown.
- Buttons can have labels or bitmaps stored in the button stack.
- The user can select any of the buttons in the matrix, thus activating the commands stored under the button.
- After activation of a button of a button stack, the stack shows the label or bitmap of this most recently activated button.

The generic Component or C_UDE dialog

Two generic dialogs for creating, deleting and modifying C_UDEs and D_UDEs are at the core of TKC. Normally, when developing new toolkits, programmers will not spend much time in tuning and tweaking a dialog of yet another entity. This is the main reason why TKC dialogs are created in a generic way. As only two dialogs exist for managing any C_UDE and D_UDE, much effort was invested in optimizing these dialogs. Moreover, different clones of the dialogs can be opened in parallel. Thus, users can manage different types of UDEs (or even the same UDE) simultaneously.

Quick help is available whenever possible. While using the dialogs, many of features will already be clear from the quick help while hovering over the respective area.

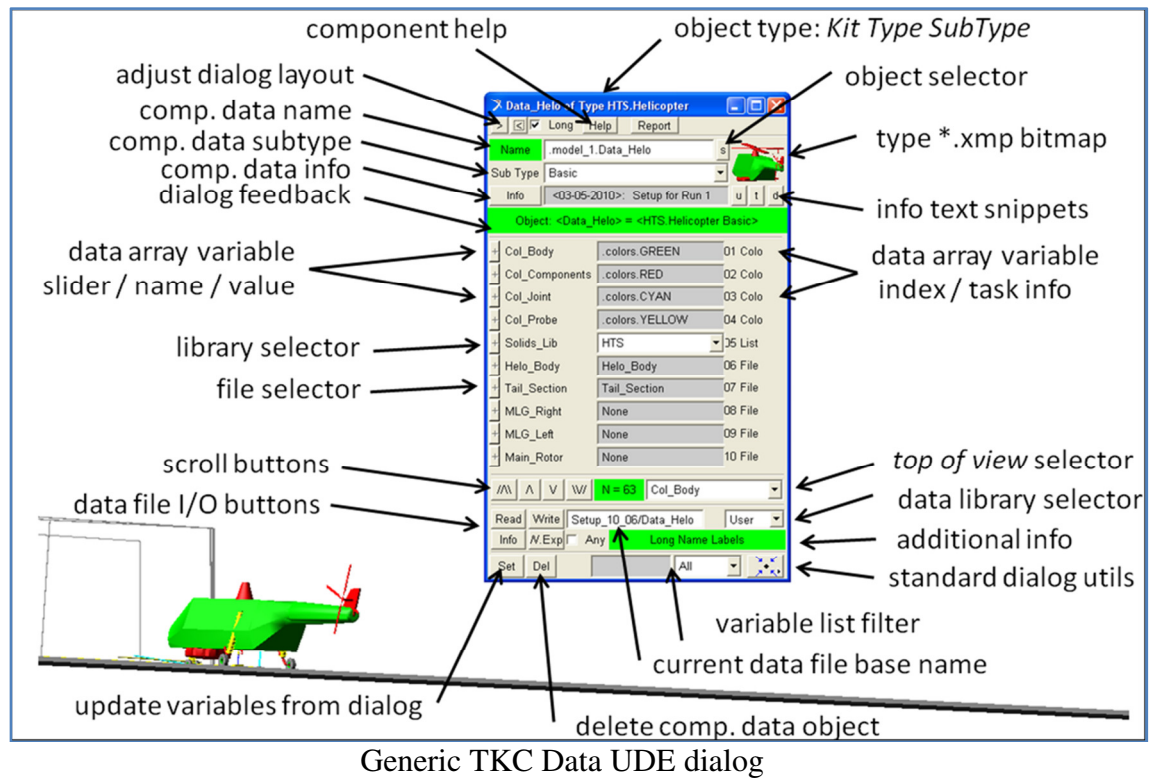


The different objects in the dialog will be discussed briefly:

- **Title Bar** Dialog header adjusts to name of C_UDE type to be defined.
- **Help** Normal on-line help on the toolkit is available via F1, the help button opens up help on the dialog itself, about TKC and the toolkit structure,
- **Bitmap** If a bitmap of the C_UDE type is defined, it will be shown here,
- **Info Buttons** Used to paste **user**, **time** or **date** info to the **Info field**
- **Info Labels** For printing information on the creation process. Color indicates new (yellow) or existing objects (green),
- **Comp. Data** The Data button can be pressed to open the D_UDE dialog to create or use a data UDE instance. If the D_UDE dialog is used to change the sub-type, the **Title Bar** will reflect this change,
- **Name Field** Input field for the name of the new or existing C_UDE. Use the **s** button to select an existing C_UDE.
- **Input Labels** The labels of parameters reflect the parameters base names,
- **Set Button** In case of a new C_UDE, the label of this button is *Create*, for an existing C_UDE the text is changed to *Set*,
- **Del Button** Deletes the C_UDE. If provided, a *.mac_del deletion macro will be called prior to deleting all C_UDE children.

The generic Component Data of D_UDE dialog

Besides the C_UDE dialog also a generic D_UDE dialog is available in TKC. Similar remarks apply with respect to the applicability and cloning features of this dialog.



Typical aspects to consider using UDEs and macros

- Some kind of version control must be used (which version of a macro works with other macros). In TKC, ADAMS models can be stored in assembly files containing only creation calls of UDE components and data components.
- Users tend to hesitate from defining extra model components in a macro-based model. While the macro part of the model can easily be re-created, the additional model components are much harder to re-create in an updated model. In TKC, the complete model can be exported automatically to an assembly file. This file is written in the ADAMS cmd language and is actually an ADAMS macro. The Non-UDE model components can either be hand-added to the assembly file or can be added in more or less automated methods. Depending on the number of objects defined different methods for capturing this information can be applied.

Placeholders used in this documentation

The following placeholders are used in the description of the C_UDEs and D_UDEs:

- **[..]** Represents a placeholder. Typically used as an input parameter of a UDE;
- **[U]** Placeholder to define the name of the UDE entity;
- **[D]** Placeholder to define the name of the UDE Data entity;
- **[M]** Placeholder to define the name of the model, in most cases the model name has been removed for readability purposes;
- **LRT** shortcut to define the (often used) `Loc_Relative_To(...)` expression;
- **ORT** shortcut to define the (often used) `Ori_Relative_To(...)` expression;

Creating and Using a new Toolkit Component

Introduction

In this Chapter, the creation of a new TKC component (or C_UDE) will be explained step-by-step using the example of a simple rigid motion driven manipulator arm. A number of steps will be used to illustrate the complete process. Where possible and applicable the work of the previous step will be used.

1. Creating the component as a non-parameterized model object, requiring a minimum of actions.
2. Parameterization of the component in a default ADAMS method. Different parameterization methods are possible in ADAMS. The example will also illustrate the advised method for model parameterization.
3. Defining the component as a registered TKC C_UDE with its D_UDE for data storage. By renaming objects defined in 2), existing objects will be captured into the model entity of the C_UDE. Finally, the C_UDE (*.mac_cre) and D_UDE (*.mac_dat) creation macros will be defined by automated storing of the model equations into a text editable file.
4. Once the C_UDE is registered (and tested) we can define a complete TKC assembly structure (i.e. a complete Base-Arm-Hand robot) on-the-fly and use the automated assembly extraction to create the version independent assembly macro file.

Description of the Component

The component description reads as follows:

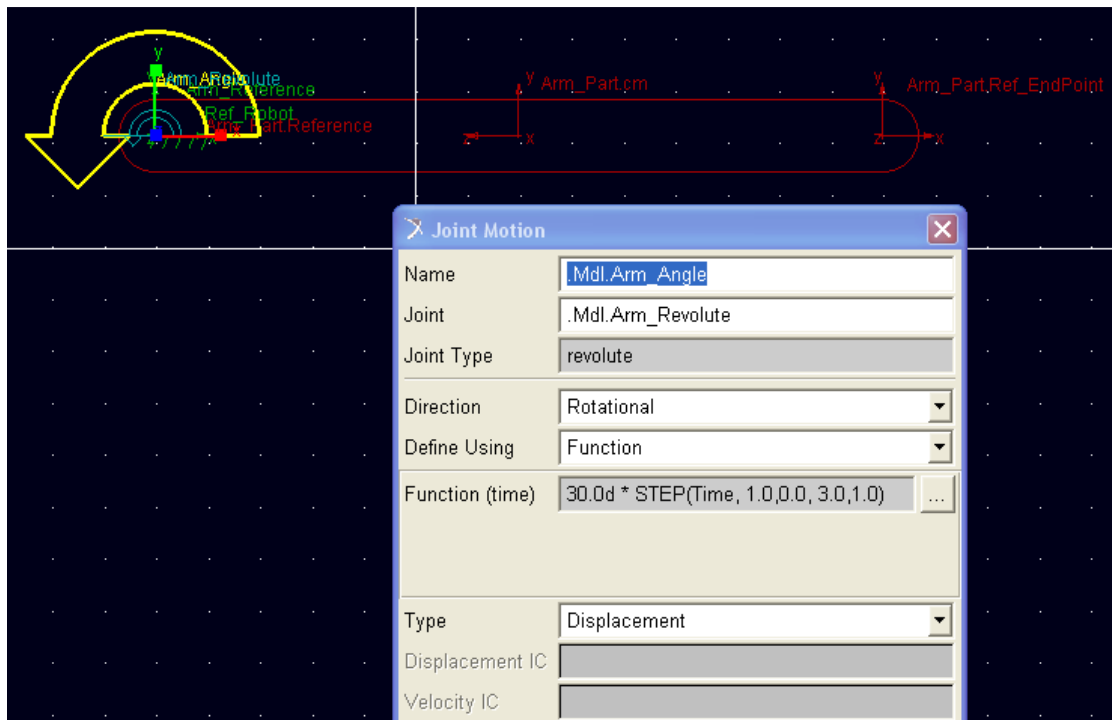
- A marker is defined on an existing *Carrier* part at the location of existing marker *Reference* under an angle about the z-axis of *Reference*.
- Part *Arm_Part* is defined as a simple rigid link geometry pointing from *Reference* to the end marker called *Ref_EndPoint*. Length, width and thickness of the link will be parameterized in a later phase.
- A Revolute joint and a Motion are defined at the origin of *Arm_Part* between *Arm_Part* and *Carrier*

Note: The pictures in this chapter were generated in an older version of TKC. Please replace all instances of *Arm_* by *Arm__* as TKC now works with a double underscore to distinguish between C_UDE object name and C_UDE children.

Building the non-parameterized component

In the first phase, preliminary naming will be used. In a later phase, when capturing the components into the C_UDE, you will rename certain components. At that point, an explanation will be given why and for what purpose this is done. Some steps in this phase may seem quite superfluous and over-complicated but they merely serve to illustrate creating the most readable and minimal component.

- Load the TKC toolkit, use setting MKS and set the grid to *Fix to View*. Rename the model to Mdl (just to match with text following)
- Create a marker on *Ground*, for convenience use orientation = view (i.e. z-axis perpendicular to the view). Name this marker *.Mdl.Ground.Ref_Robot*.
- Create a Link, starting from *Ref_Robot*, pointing to the right. Rename the new part to *Arm__Part* and the link to *Arm__Part.Shape* (just to illustrate the relative unimportance of geometric entities in ADAMS).
- Create a revolute joint with *Arm__Part* as first part and *Ground* as second part. Rename the joint to *Arm__Revolute*, and inspect three markers at the joint location. Two markers are used by *Arm__Revolute* (one on *Ground* and one on *Arm__Part*). The third marker is the I_Marker of *Arm__Part.Shape*. A powerful method for verifying this (please do so) is by retrieving the info of *Shape* and *Arm__Revolute* and/or opening the respective modify dialog.
- To obtain a clean model, we want to minimize the number of markers used. We can do this by using the I_Marker of *Arm__Revolute* also as I_Marker for *Shape*. This will be done in two steps:
 1. Rename the I_Marker of *Arm__Revolute* to *Arm__Part.Reference* and the J_Marker of *Arm__Revolute* to *Ground.Arm__Reference*
 2. Open the modify dialog for *Arm__Part.Shape* and replace the I_Marker used (probably called *Arm__Part.Marker_2*) by *Arm__Part.Reference*. Now marker *Arm__Part.Marker_2* is superfluous and can be deleted.
- Rename the J_Marker of *Arm__Part.Shape* to *Arm__Part.Ref_EndPoint*. In a later phase this marker may be used by other components, i.e. as reference marker for the next mounted manipulator arm.
- Create a motion *Arm__Angle* on *Arm__Revolute* and prescribe the function for the motion displacement as $30.0d*STEP(Time, 1.0,0.0, 3.0,1.0)$. The input angle of the Arm will change using a polynomial function from zero degrees to 30 degrees from time is one seconds to three seconds.



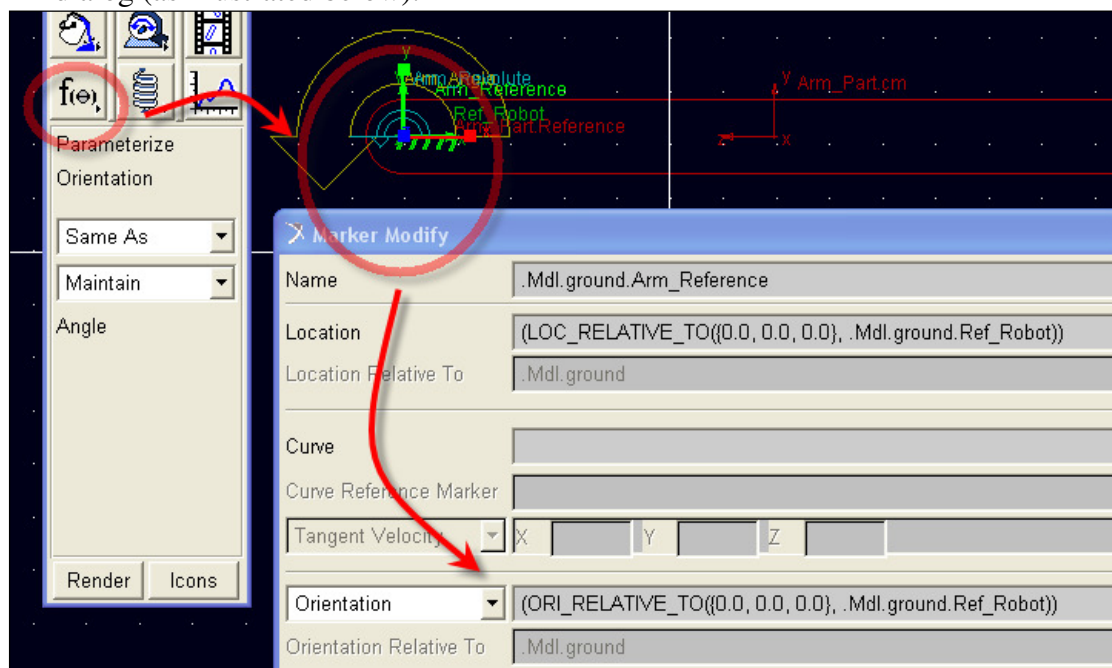
- The figure above shows an overview of the model (*Arm_Part* components in red, *Ground* markers in green, *Arm_Revolute* in Cyan and *Arm_Angle* in yellow).
- Now the arm component is ready and we can test its functionality. Therefore: run the model for 5 seconds and check if indeed we can see the arm rotate over 30 degrees counter clockwise. If these are the only components in the model, the number of Degrees of Freedom (use the Gruebler Count) must be zero.

Parameterization of the component structure

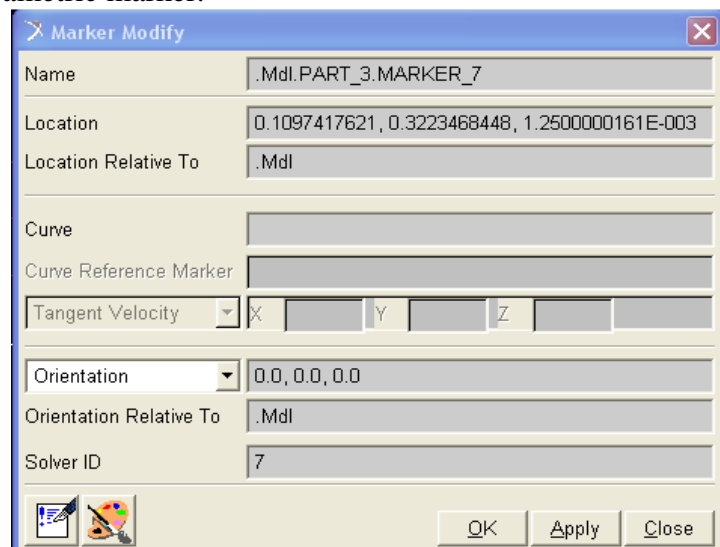
After the first phase, the component is defined in a reasonably minimal (no superfluous markers) and readable (intuitive use of object names) method. However no parameterization is applied yet, so no design study can yet be performed with it.

The following steps are required to fully parameterize the component:

- *Ground.Ref_Robot* will serve as a so-called *Master* to define the base location of *Arm_Part*. Therefore, location and orientation of *Ground.Arm_Reference* are parameterized to *Ground.Ref_Robot* using the required buttons in the main toolbox. In the figure below the steps required for parameterization of the orientation are illustrated. Especially when you are not yet experienced in the ADAMS parameterization method, check the results by opening the *Slave Marker* modify dialog (as illustrated below).

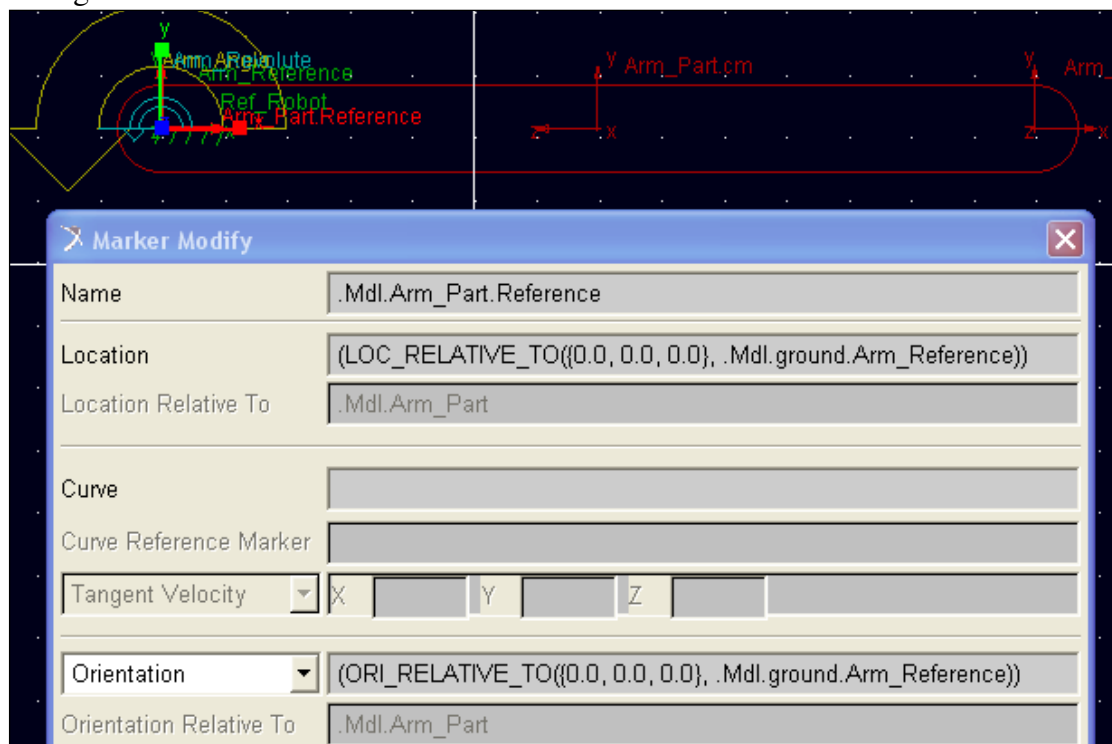


For understanding compare the definition of marker *Mdl.Ground.Arm_Reference* with a non-parametric marker.



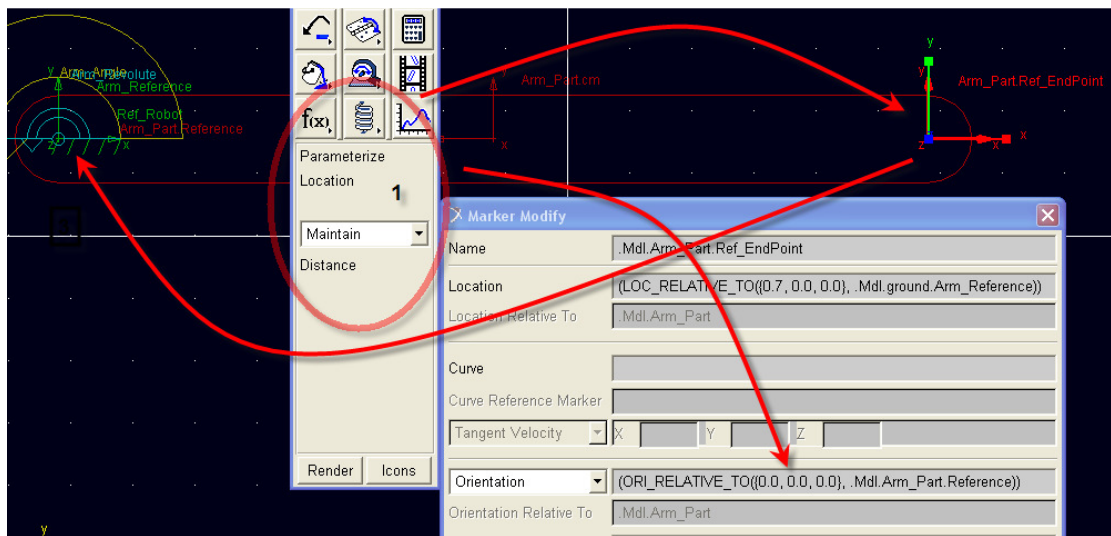
See the definition of the example marker `Mdl.Part_3.Marker_7` above. Notice which fields are not grayed out and the numbers (without brackets) in the *Location* and *Orientation* fields. By definition, the numbers defining the location and orientation of a non-parametric marker are *global* vectors and follow the defined ADAMS/View units for length and angle. As soon as field values contain text between brackets, ADAMS/View will assume an expression. By definition, ADAMS/View expressions for location and orientation produce *local* vector variables.

- Marker `Ground.Arm__Reference` will serve as local reference for `Arm__Part` and its components. Therefore, in a similar approach as above, `Arm__Part.Reference` will be made a *Slave* to `Ground.Arm_Reference`. The result is illustrated in the figure below.

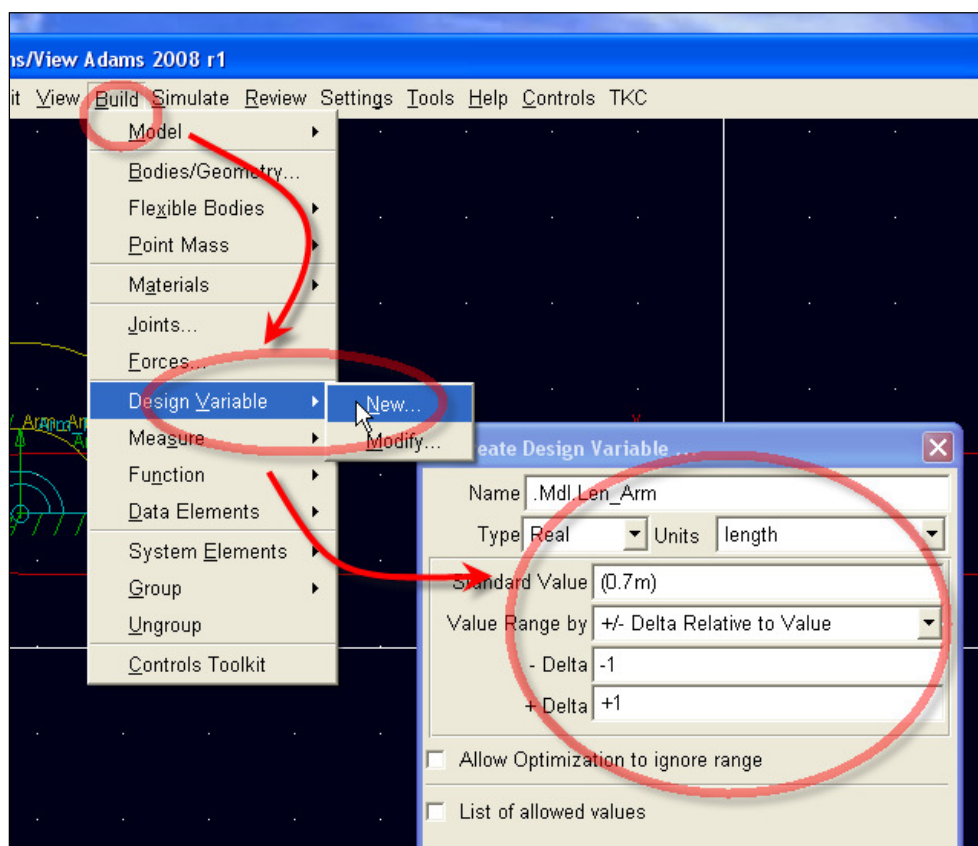


- Effectively, marker `Ground.Arm__Reference` serves as global origin and marker `Arm__Part.Reference` as local (part-owned) origin for part `Arm__Part`. Working in this sequence, and using `Arm__Part.Reference` where possible as origin for objects in `Arm__Part` is the advised method for parameterization in ADAMS but is also an intuitive top-down method in general in a system approach to define a structured parameterization of model components. As an example of the advantage to use a part-owned origin: a copy of `Arm__Part` will result in a new part `Arm__Part_2` including all its child markers and geometry. All one has to do to fix it to another global position is to make `Arm__Part_2.Reference` a slave to the new global reference.
- Continuing the process we will now make `Arm__Part.Ref_EndPoint` a slave to `Arm__Part.Reference`. At this point we will define the length of the link with the design variable `.Mdl.Len_Arm`. The required basic steps are illustrated:
 - Use the current length of the link and make `Arm__Part.Ref_EndPoint` a slave to `Arm__Part.Reference` use option *Maintain* in the *Main Dialog Lower window* to freeze to the current value of the relative distance. In the

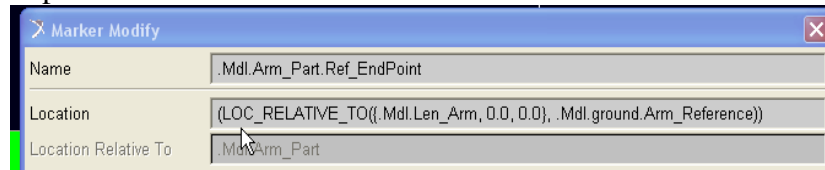
illustration below the distance between these markers appears to be 0.7 meters. As a result, *Ref_EndPoint* is defined in the frame of *Arm_Part.Reference* at vector $(\{0.7,0.0,0.0\})$ m



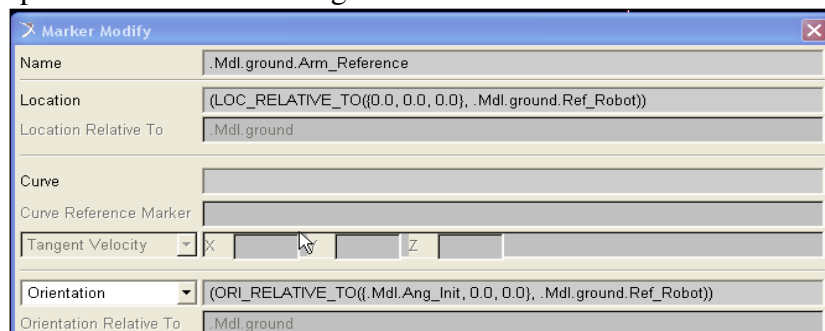
- Define design variable *.Mdl.Len_Arm* as illustrated below. It is strongly advised to use units=length. Defining the type of length units (in this example *Standard Value = (0.7m)*) is not required but may increase the readability of the model settings. To avoid confusion we will define the value equal to the value found in the parameterization of *Ref_EndPoint*.



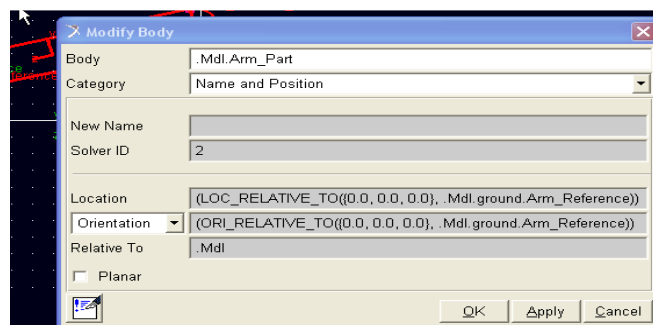
- Now use *Len_Arm* in the expression for the location of *Ref_EndPoint* . This can be done using the Expression builder, or simply by text editing the expression.



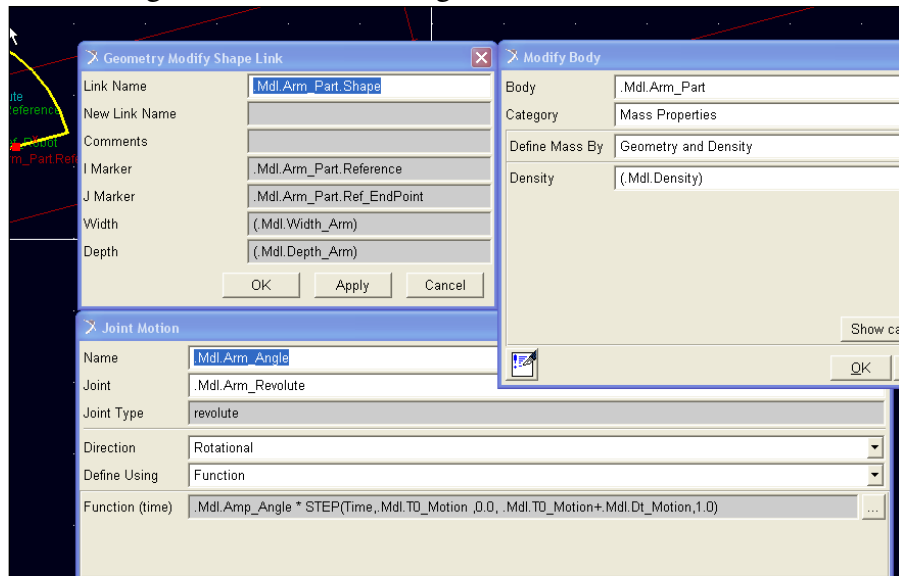
- You can check the expression by changing the value of *Len_Arm*, if all is correct, the length of the link must change appropriately.
- In the same way we will define design *Width_Arm* and *Depth_Arm* for the size of the link
- Variable *Density* will be used to define the total mass and inertia of *Arm_Part*.
- Variables *TO_Motion*, *Dt_Motion* and *Amp_Angle* are defined to parameterize the Run-Time function for motion *Arm_Angle*.
- Finally, the component kinematics is completed with initial angle *Ang_Init* of *Ground.Arm_Reference* relative to *Ground.Ref_Robot*. Notice that, as ADAMS uses Z-X-Z Euler angles for transformations, *Ang_Init* is the first component of the relative angle vector.



- The last phase of the geometric parameterization of *Arm_Part* is also the most cryptic: we will parameterize the local part reference frame (*lprf*) to coincide with the location and orientation of marker *Ground.Arm_Reference*. Although is not mandatory to do so, there is a number of good reasons to keep the *lprf* close to the cg of a part. Another aspect to note is that parameterization of the *lprf* is usually the last operation to perform; doing so before parameterizing part child geometry may result in a geometry mess. The syntax to make the *Arm_Part* *lprf* a slave to *Ground.Arm_Reference* is listed below.



- The usage of the different design variables is illustrated below.



- Below, the ADAMS Table Editor tool is used to illustrate the definition of the different design variables mentioned.

Table Editor for Real Variables in .Mdl

Amplitude of Arm Motion Angle

	Real_Value	Units	Comments
Amp_Angle	(30.0d)	"angle"	"Amplitude of Arm Motion Angle"
Ang_Init	(15.0d)	"angle"	"Arm Initial Angle"
Density	(7801.0(kg/meter**3))	"density"	"Density of Arm Part"
Depth_Arm	(5.0E-002m)	"length"	"Depth of Arm Link Shape"
Dt_Motion	(2.0sec)	"time"	"Start Time of Arm Motion Angle"
Len_Arm	(0.7m)	"length"	"Length of Arm Link Shape"
TO_Motion	(1.0sec)	"time"	"Rampup Time of Arm Motion Angle"
Width_Arm	(0.1m)	"length"	"Width of Arm Link Shape"

Parts Markers Points Joints Forces Motions **Variables**

At this point, the component geometry, inertia and kinematics are fully parameterized. Check and verify correctness by changing variables and visually checking and/or performing simulations. For the model at hand, simulations must be performed without any warnings at any step in the simulation process. Once this has been verified, the model is considered as successfully parameterized.

Register the Component as a TKC C_UDE

Up to this point, all actions performed have been standard ADAMS. We may have used some of the functionality in TKC (such as grid and/or icons displaying shortcuts) but we did not create anything non-standard ADAMS. By registering the robot arm component as a TKC component and its parameters as TKC data array we will extend the (re-)usability of it.

The method for finally performing the registration of the component can vary from:

- A completely text based approach, i.e. editing all required ASCII files and incrementally reloading the mother toolkit until all functionality is correct.
- To an almost completely GUI driven approach where model components are extracted from the model into macros for the component to be defined.

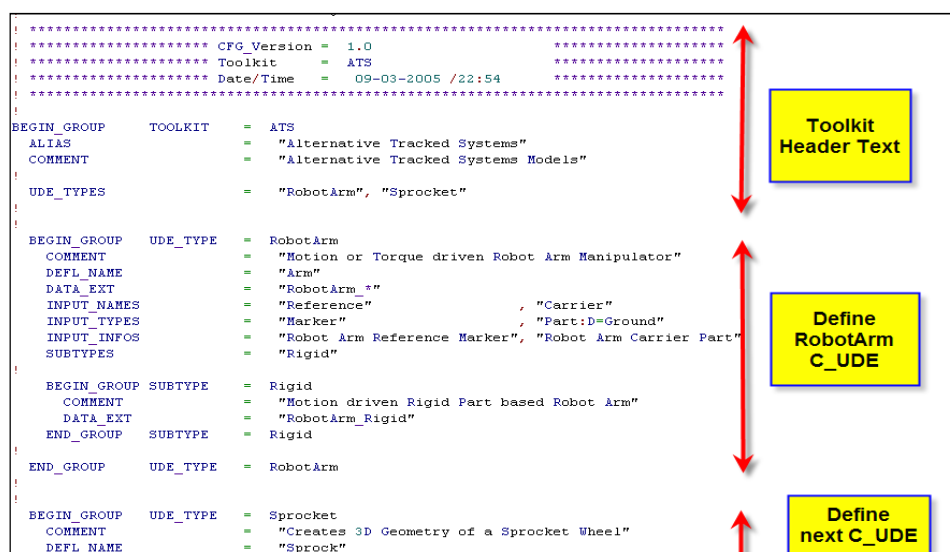
In this example we will use a mixture of both. The explanation will be on ad-hoc basis, mainly focusing on the principle of the process and not on details of the implementation.

In publishing the component to TKC the following actions are performed:

1. Definition of the interface of the component so TKC generic dialogs will be displayed with the correct number and type of input parameters (see marker *Reference* and part *Carrier*). This can also include additional actions that must be performed when creating, modifying or deleting this component.
2. Creation of the macro required to create an instance of the component (C_UDE) in the active model and a data macro to create an instance of the data array (D_UDE) required by these C_UDEs. The syntax of these macros will be shown for the example of the Robot Arm.

Registration of the Component and its Interface

For step 1. we will use text editing to add the *RobotArm* to an existing Toolkit. The required lines will be copied from an existing component and adjusted where needed. The final definition for the new *RobotArm* component is included in the figure below, which is extracted from the configuration file *_ATS.cfg* of the *ATS* toolkit.



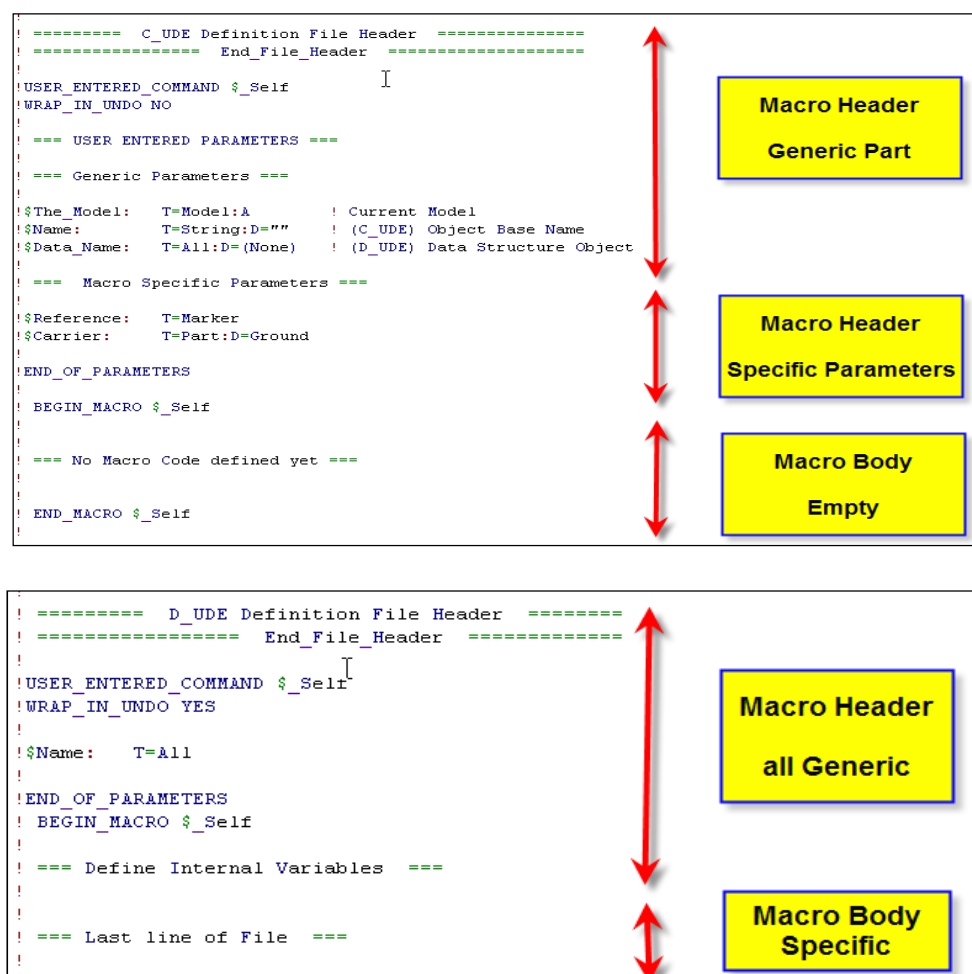
Implementing the functionality in this cfg file is done by reloading the toolkit which is done using the TKC Control Dialog. The figure below lists the steps for

1. Opening the TKC Control Dialog,
2. selecting the Toolkit to be Refreshed and Reloaded and
3. Refreshing and Reloading ATS

Basically refreshing means that all binary stored data for ATS is deleted which is required when new toolkit functionality must be available in a session. When all syntax rules are followed correctly, the ATS Toolkit Dialog will be displayed with button stacks for the different available components in ATS. The functionality of the ATS dedicated Dialog is comparable to that of the ADAMS Main Toolbox.

Definition of the Component Creation Macros

As mentioned, all syntax errors in the cfg file and/or in the available macro template are reported in the TKC session. For the *RobotArm* two macro templates must exist: *RobotArm_Rigid.mac_cre* and *RobotArm_Rigid.mac_dat* which can be verified from *_ATS.cfg* as *RobotArm* only defines the subtype *Rigid*. Both macros fully apply to the ADAMS/View command language syntax, and the templates are listed below.



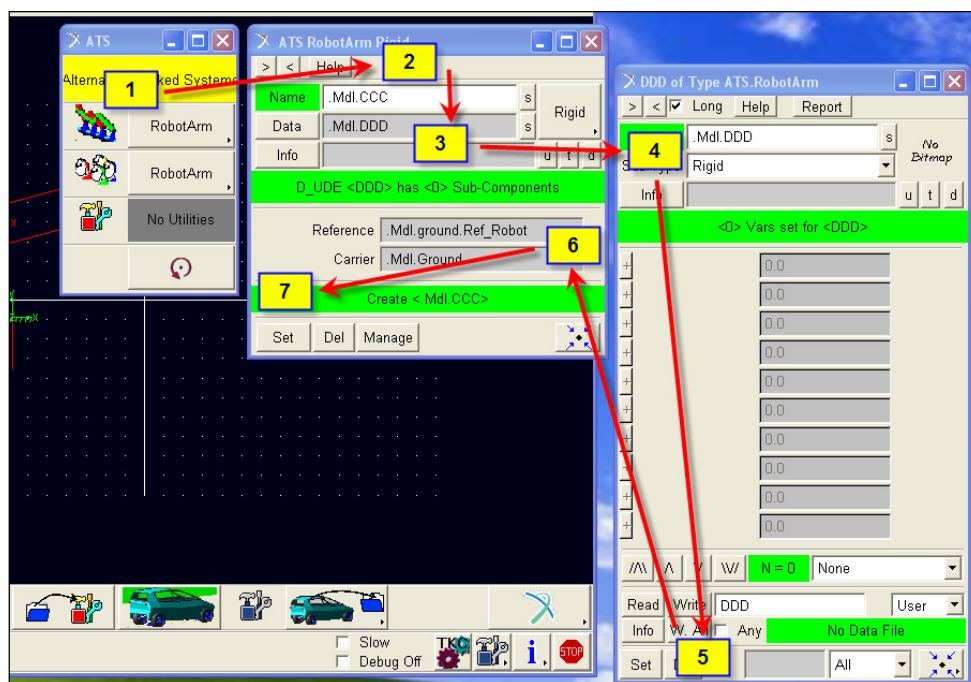
The following rules apply to the macros:

- The **.mac_cre* macros have generic and specific parameters in a strict sequence:

1. *The_Model*: Name of the current model,
 2. *Name*: Text string with the base name of the C_UDE to be defined,
 3. *Data_Name*: Name of the D_UDE model object to be used by the C_UDE.
 4. ... The remaining parameters of the macro are the specific input parameters for the C_UDE, they must be listed in the *INPUT_NAMES* definition line in the toolkit cfg file. For the *RobotArm* these inputs are *Reference* and *Carrier*.
- The *.mac_dat macros typically only have one parameters representing the name of the D_UDE model object (defined as a parent design variable). A completed *.mac_dat macro will typically contain definition lines for all design variables in the D_UDE array structure.

Now we have defined *RobotArm* as a new (but completely empty) component of the toolkit ATS without any component data whatsoever. We will add contents (i.e. code) to *RobotArm* by encapsulating the existing parameterized model object into a *RobotArm* model instance and storing this as the new definition of a *Rigid RobotArm* structure and data for a *Rigid RobotArm*.

The encapsulation starts by defining an empty *RobotArm*. To avoid naming errors, as the model code extraction is based on object names we will define a D_UDE named *DDD* and a C_UDE named *CCC* (of course you may also use other *unlikely* names).



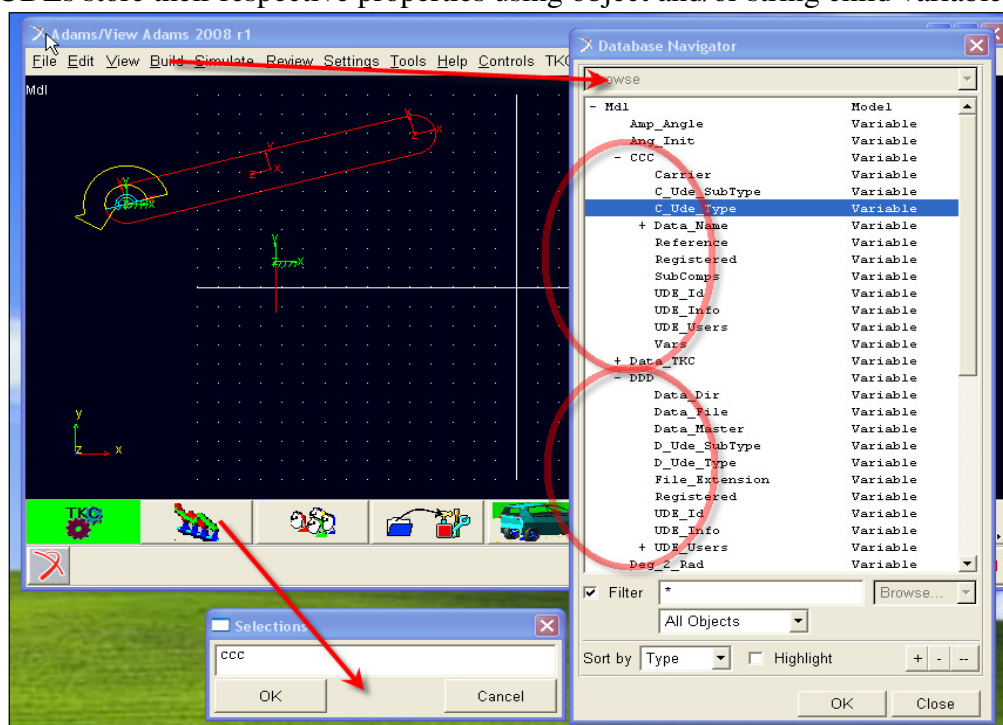
The step-by step creation of *DDD* and *CCC* is shown using the figure above:

1. Press the button on the *ATS* dialog to open the generic C_UDE dialog,
2. Enter the names for *CCC* and *DDD* to be defined,
3. Press the *Data* button to open the generic D_UDE Dialog,
4. Verify the right Sub-Component type, in this case only *Rigid* exists,
5. Create the D_UDE *DDD*, the labels on the D_UDE dialog will change, here yellow means a new object and green means an existing object,

6. Complete the input fields for *CCC*. To encapsulate the existing arm, use the same inputs: *Reference* = *.Mdl.Ground.Ref_Robot* and *Carrier* = *.Mdl.Ground*,
7. Create the *C_UDE CCC*.

Both on the *C_UDE* and on the *D_UDE* the label on the *Create* button will switch to *set* to denote that the object indeed exists and can be updated when pressing this button (in general this is not used often).

Next, the figure below shows the presence of *.Mdl.CCC* and *.Mdl.DDD*. Both objects can be edited (i.e. the *C_UDE* or *D_UDE* dialog will be opened) when clicking the respective buttons on the bottom main toolbar. A tree view of objects in *CCC* and *DDD* is listed in a selective view of the database navigator on all design variables (*Build*, *Design Variable*, *Modify*) in *Mdl*. From the child variables of *CCC* and *DDD* we can extract some of the functionality of *TKC*. Basically, *TKC C_UEs* and *D_UEs* store their respective properties using object and/or string child variables.



The final publication of the desired macro code is done in the following steps:

1. Renaming objects to become part of *CCC*.

The following renames are done:

- *.Mdl.Arm__Part* → *.Mdl.CCC__Part*
- *.Mdl.Arm__Revolute* → *.Mdl.CCC__Revolute*
- *.Mdl.Arm__Angle* → *.Mdl.CCC__Angle*
- *.Mdl.Ground.Arm__Reference* → *.Mdl.Ground.CCC__Reference*

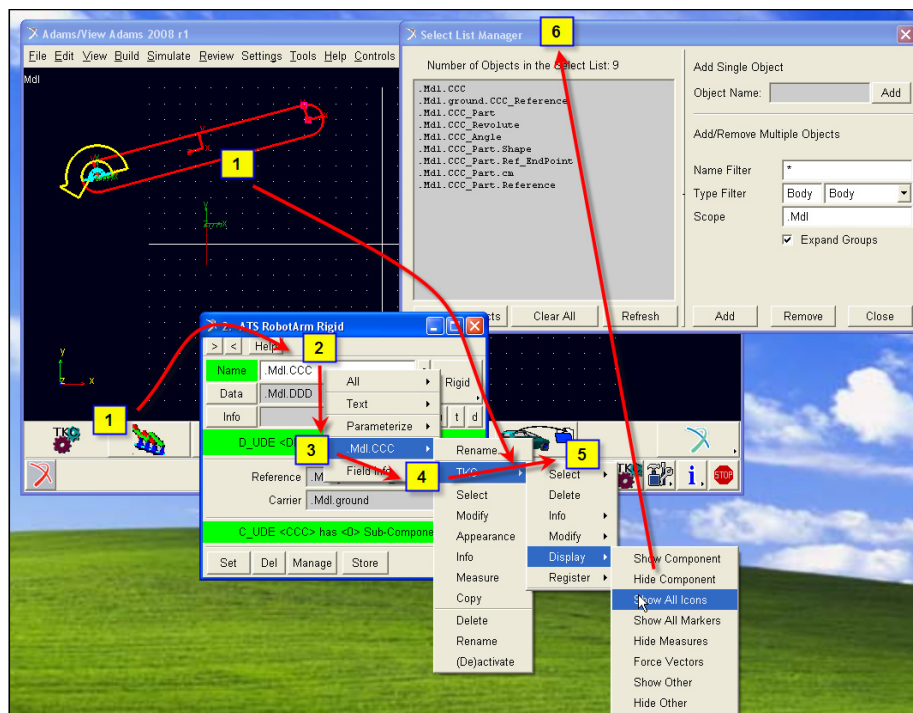
Although this procedure may seem a bit cryptic, actually it is very structured:

- All objects belonging to *CCC* must have a base name starting with the string *CCC__*.

- For all model objects in the top level of a model, such as parts constraints, motions, forces and measures this is the string following the model name.
 - For markers and geometries, or in general children to parts, either the base name must start with *CCC__* (in case the parent part is not inside the *C_UDE*) or the part name starts with *CCC__*.
2. Verify if the encapsulation of Model objects into *CCC* is complete and successful. A dedicated TKC button is defined in the main Object Pulldown. Using this button one can manipulate any TKC-based model objects. The procedure to access this functionality is shown below.

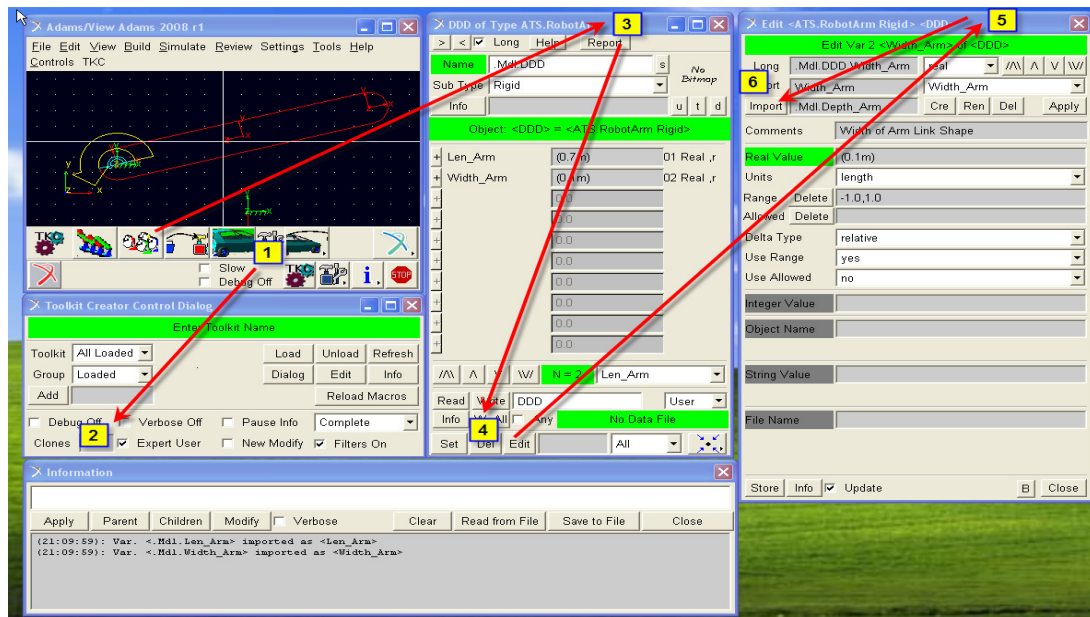
The figure below shows two general methods to manipulate a TKC *C_UDE*:

- a) by opening the *C_UDE* modify dialog (lower 1), making a right mouse click in the *Name* field (2), doing a pull down walk-through until the *C_UDE* object can be hidden or displayed or manipulated in another way (3, 4, 5).
- b) by right clicking on an object in the model view belonging to a *C_UDE* (upper 1), walking through the pull down menus one will obtain the same function (4, 5) .



Notice that, as in ADAMS standard, many of the TKC manipulations are using the *Select List*. Thus, opening the Select List Manager (6) will show all objects contained in *C_UDE CCC*.

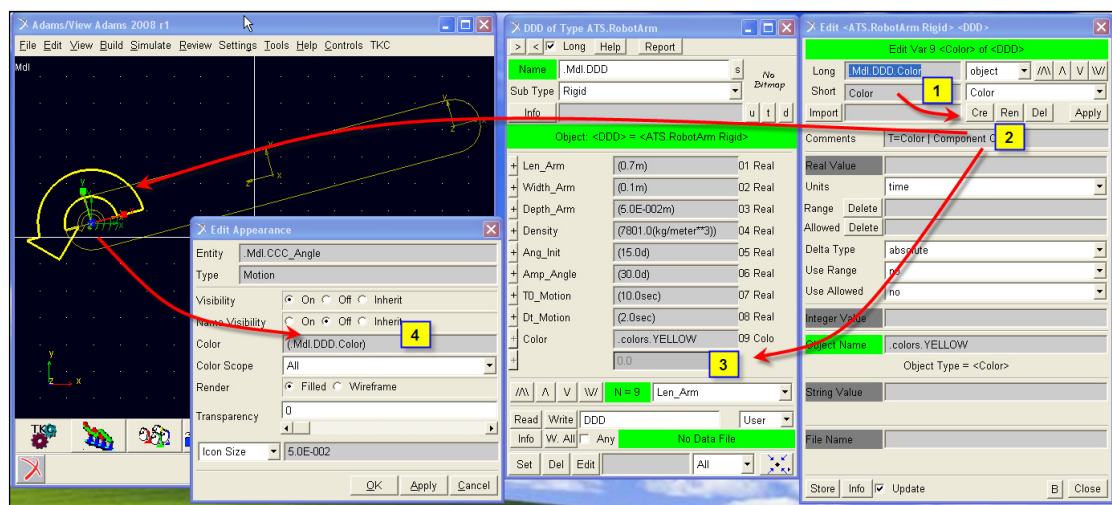
Now all model objects required for the Robot Arm have been encapsulated into *CCC* and will be treated as any other TKC *C_UDE* child object. Next we will encapsulate all necessary design variables into *DDD*. Effectively, in the ADAMS session we will use the Edit *D_UDE* Dialog and import the design variables into *DDD*. The required actions are shown in the figure below:



1. Open the TKC Control Dialog to
2. Switch to Expert User Mode to show the *Edit* button on the D_UDE dialog,
3. Open the D_UDE Dialog for *DDD* and
4. click on the Edit button to edit the design variable children of *DDD*,
5. In the Edit D_UDE Dialog variables can be listed, added, deleted and type and comments can be modified,
6. Click on *Import* and browse for the variables used by *CCC*, after successful import, the D_UDE Dialog will directly report the variables as children of *DDD*.

Effectively, variable *.Mdl.Len_Arm* will be made a child of *DDD* by renaming it to *.Mdl.DDD.Len_Arm*. All properties of the design variable will be maintained and can be edited at will.

Also you can add additional design variables to complete the *RobotArm*. For example we will add variable *.Mdl.DDD.Color* to define the color of a certain instance of the *RobotArm*. The steps for defining and using this color variable are explained below.

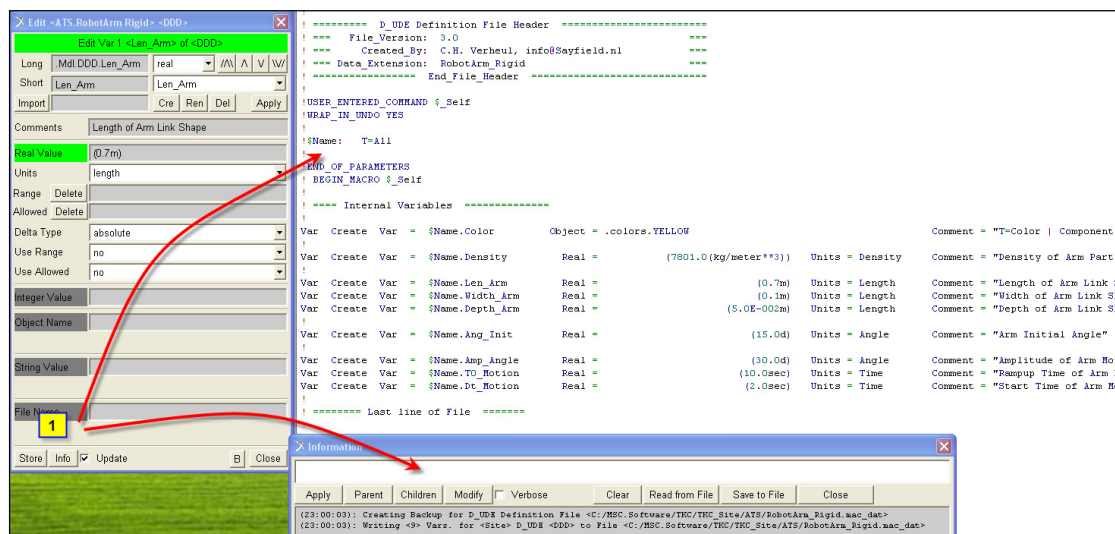


1. In the Edit D_UDE Dialog Type *Color* in the *Short Name* Field. Set the type selector to object and enter the object value *.Colors.Yellow* to define the color

properties. For extra documentation you may add a leading string “*T=Color |*” to the variable *Comments* value.

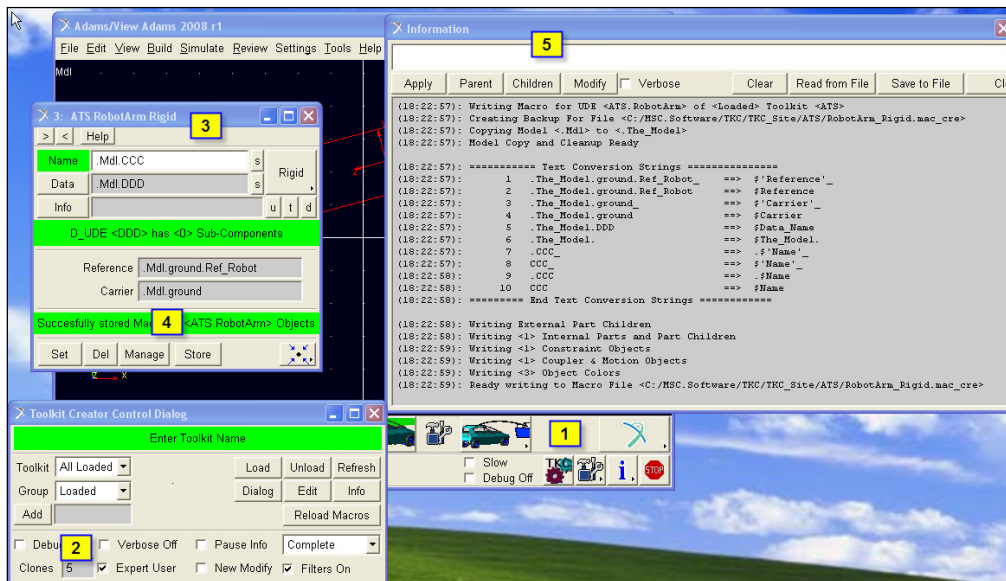
2. Press the *Cre* button to finally create the color variable,
3. As the *Update* toggle button is true, automatically the D_UDE Dialog is updated with a line with properties for *.Mdl.DDD.Color*,
4. The object variable *.Mdl.DDD.Color* can now be applied to the objects in *CCC* by right clicking on an object and opening its Appearance Dialog. The name of the Color can be copied and pasted from the *Long* Name field value in the Edit D_UDE Dialog.

Formatted storage of the D_UDE data structure to a *.mac_dat file is performed by pressing the *Store* button in the Edit D_UDE dialog. The sequence of the variables in the data file can be changed by hand editing (i.e. to obtain a better grouping of variables). The info panel displays messages about the name of the *.mac_dat file and the number of variables stored.



At this point, the C_UDE equations for the Robot Arm can be exported to the macro file. This requires the following steps:

1. Open the Toolkit Creator Control Dialog,
2. Switch to TKC Expert User Mode,
3. In Expert User Mode, open the C_UDE Dialog which now shows an additional *Store* button



4. Click the *Store* button to activate the macro export.
5. The Info dialog reports textual info on the export process. The info lines indicate the place holders used and the number of model objects exported to the macro.

The formatted text in the body of the C_UDE creation macro is listed below.

Please replace all instances of '\$Name' by '\$Name'__

```

!
! ===== UDE External Part Child Objects =====
!
Marker Create Marker = $Carrier.$Name.Reference &
Loc = (LOC_RELATIVE_TO((0.0, 0.0, 0.0), $Reference)) &
Ori = (ORI_RELATIVE_TO(($Data_Name.Ang_Init, 0.0, 0.0), $Reference))
!
! === Define Part ===
!
Part Create Rigid Name Part = $The_Model.$Name.Part &
Loc = (LOC_RELATIVE_TO((0.0, 0.0, 0.0), $Carrier.$Name.Reference)) &
Ori = (ORI_RELATIVE_TO((0.0, 0.0, 0.0), $Carrier.$Name.Reference))
!
Marker Create Marker = $The_Model.$Name.Part.Reference &
Loc = (LOC_RELATIVE_TO((0.0, 0.0, 0.0), $Carrier.$Name.Reference)) &
Ori = (ORI_RELATIVE_TO((0.0, 0.0, 0.0), $Carrier.$Name.Reference))
!
Marker Create Marker = $The_Model.$Name.Part.cm &
Loc = 0.0, 0.0, 0.0 &
Ori = 0.0, 0.0, 0.0
!
Marker Create Marker = $The_Model.$Name.Part.Ref_EndPoint &
Loc = (LOC_RELATIVE_TO(($Data_Name.Len_Arm, 0.0, 0.0), $The_Model.$Name.Part.Reference)) &
Ori = (ORI_RELATIVE_TO((0.0, 0.0, 0.0), $The_Model.$Name.Part.Reference))
!
! ===== Inertia and geometries for Part =====
!
Part Create Rigid Mass Part = $The_Model.$Name.Part Density = ($Data_Name.Density)
!
Geometry Create Shape Link Link = $The_Model.$Name.Part.Shape &
I_Marker = $The_Model.$Name.Part.Reference &
J_Marker = $The_Model.$Name.Part.Ref_Endpoint &
Width = ($Data_Name.Width_Arm) &
Depth = ($Data_Name.Depth_Arm)
!
! ===== Constraint and Motion Objects =====
!
Constraint Create Joint Revolute Joint = $The_Model.$Name.Revolute &
I_Marker = $The_Model.$Name.Part.Reference &
J_Marker = $Carrier.$Name.Reference
!
Constraint Create Motion Motion = $The_Model.$Name.Angle &
Joint = $The_Model.$Name.Revolute &
Type = Rotational &
Function = "($Data_Name.Amp_Angle * STEP(Time,$Data_Name.TO_Motion,0.0, $Data_Name.TO_Motion+$Data_Name.Dt_Motion,1.0))" &
Time_Der = Displacement
!
! ===== Object Colors =====
!
Entity Attributes Entity Name = $The_Model.$Name.Part Color = ($Data_Name.Color)
Entity Attributes Entity Name = $The_Model.$Name.Revolute Color = ($Data_Name.Color)
Entity Attributes Entity Name = $The_Model.$Name.Angle Color = ($Data_Name.Color)
!
! ===== Last line of File =====
!

```

Data_Name
Carrier
Reference
Name

After some text simplifications, the relevant parameters of the macro are color highlighted to illustrate their use for the definition of objects in the component.