TTTech

Ensuring Reliable Networks
TTTech Computertechnik AG

TTP Plan

The Cluster Design Tool for TTP

Product version 9.0.18 of 31-May-2024
Manual edition 9.0.18 of 31-May-2024
Document number D-001-G-01-003

Contents

1. Introduction

This document describes TTTech's cluster design tool TTP-Plan. TTP-Plan is a comprehensive tool for the design of TTP®clusters based on the concepts of state messages and temporal firewalls.

A cluster contains a number of nodes exchanging messages in a statically defined temporal pattern. These nodes are organized into one or more *webs* (see Figure 1). A web is a set of nodes connected by an independent physical medium, e.g., one bus, over which communication takes place. All webs of a cluster are assumed to be synchronized (precision PI) to each other.

If designed correctly, webs can improve failure safety in a similar way as multiple channels (i.e., more bandwidth or redundancy). If required, each web can use a different cable type (optical or electrical), which enhances the resistance against various external sources of interferences. This also provides the possibility to use the cheaper electrical cable for webs with a lower required transmission speed, thus saving cost.

TTP-Plan models all webs at the cluster level, whereas TTP-Build models them at the node level.

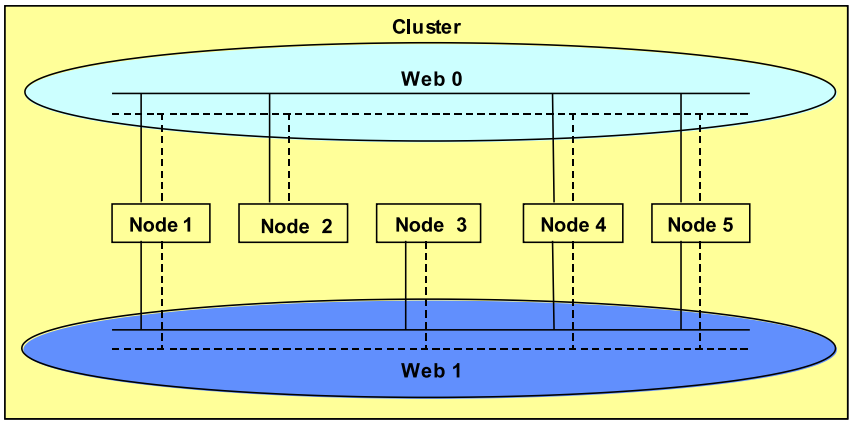


Figure 1: Example of a cluster with two webs

TTP-Plan requires the precise definition of communication patterns on the bus. Thus, the interfaces between the nodes of the cluster are completely defined in both the value and time domain. TTP-Plan is designed to be platform-independent and object oriented. A 'Pilot' window supports direct navigation through the essential object model of the cluster. The cluster is designed by specifying the relevant objects and the relations between them, i.e., the nodes and messages.[1]

Once the cluster has been designed, TTP-Plan checks the design for correctness at all levels (object, global and schedule). All errors are displayed in a special error browser providing detailed information. In many cases a simple click on an error message directly opens the appropriate editor where the error can be corrected.

Once the design is consistent and complete, TTP-Plan can generate a cluster schedule. This is either done from scratch, or by extending an existing schedule (*schedule extension*). During schedule extension the scheduler adds new messages to frames without affecting the nodes that do not use these messages.

TTP-Plan provides a schedule editor to inspect and modify the generated schedule. Manual modification can be used in cases where the scheduler fails to find a feasible schedule, or to attain a different allocation of messages to frames.

From TTP-Plan's specification of the bus communication, data structures called message descriptor lists (MEDLs) can be generated (by TTP-Build) and loaded into the controllers of the cluster nodes (with TTP-Load).

This has the following advantages:

- All nodes get consistent information from a common source.
- For minor updates, only the application needs to be downloaded, while the MEDL can remain unchanged.

Besides the direct manipulation graphical interface, the cluster design can also be defined or modified using a scripting interface (Python language). Searching for objects and renaming them is often more conveniently done in a textual environment. The programming interface of TTP-Plan provides the possibility of querying and changing the cluster design and of passing cluster information to other tools.

1.1. The TTP-Tools Software Development Suite

TTP-Plan is part of TTTech's TTP-Tools software development suite. This tool suite is designed to fully exploit the advantages of the Time-Triggered Architecture (TTA). It provides an innovative two-level design framework, which supports the seamless integration of heterogeneous electronic subsystems, developed by different suppliers, into an integrated computer system. This requires a composable network architecture that differentiates clearly between system and subsystem issues. To this aim, the TTP-Tools support the precise specification of the temporal and functional interfaces between the subsystems of a TTA.

[1] Plus information about the messages' data types and packaging.

At the system or cluster level, the system integrator defines the subsystem functions and specifies the communication interfaces in the value and time domains precisely. At the subsystem or node level, the component supplier retains complete control over all hardware and software design decisions, as long as he complies with these interfaces. (Of course, with the restriction that this one supplier supplies all subsystems for that node; if he only provides part of them, he does not have this kind of control.) The unambiguous interface specification enhances ease of integration, quality, and reusability of the developed products. The composability of the architecture prevents the occurrence of unintended (adverse) effects during system integration.

The "separation of concerns" inherent to TTTech's two-level design framework translates directly into significant business benefits for all parties involved. The clear definition of responsibilities prevents the omission of essential functions as well as the duplication of efforts resulting in waste of time and potentially conflicting implementations. For both the system integrator and the subsystem suppliers, delimitation of responsibilities restricts the potential for conflicts and reduces the communication overhead.

For the component supplier, the two-level design framework provides a high degree of independence. The supplier is unrestricted in his design choices, needs less system information, and benefits from a substantial decrease in requirements changes. For the system integrator, the two-level design framework offers all the benefits of composability: the costs, time, and risk of system integration are reduced and there is less need for expensive testing later on to achieve temporal composability, if it has already been part of the design from the beginning.

By supporting faster and more precisely estimable turn-around times, this approach permits a shorter time-to-market of TTP-based products.

Figure 2 shows the interactions of the TTP-Tools according to the two-level design:

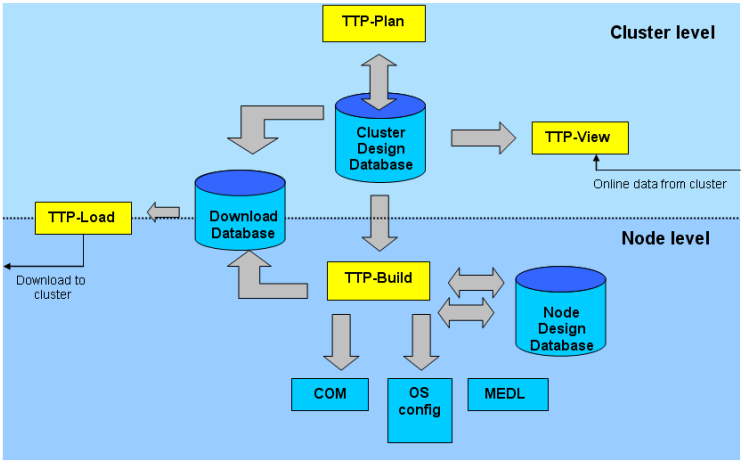


Figure 2: The TTP software development environment

- At the cluster level, the cluster design tool TTP-Plan provides information via the *cluster database (CDB)* to the node design tool TTP-Build, to the monitoring tool TTP-View, and to the download tool TTP-Load. In TTP-Plan you specify the communication patterns for the exchange of messages between the nodes on the bus. From this, a complete communication schedule (also including the timing information) is created automatically. As a result, the cluster design serves as a central database for subsequent node designs, monitoring, and other purposes. TTP-View allows on-line visualization of the traffic on a PC. TTP-View collects all information via a Monitoring Node, which is tightly synchronized with the cluster and monitors all real-time activities. The Ethernet interface of the Monitoring Node is connected to the PC running TTP-Plan via a standard Ethernet connection. This decoupling of cluster and monitoring system guarantees consistent representation of the data sent on the bus.
- At the node level, the software development suite includes TTP-Build and TTP-Load. TTP-Build supports the design of the application software for a single node of a cluster. Based on this information, TTP-Build generates the communication layer (COM) and configures the operating system on the node. TTP-Build hence provides the interface between the cluster level (i.e., the cluster schedule) and the node level (i.e., the application software).

  TTP-Load is TTTech's download tool. It transfers the MEDLs – generated by TTP-Build – and the user-defined applications to the appropriate nodes in the cluster.

1.2. Section Overview of the User Manual

The contents of the user manual are organized as follows:

- The tutorial (Section 2) provides a step-by-step guide that takes you through the design of a cluster and its subsystems, messages, message types, scheduling, and schedule editing. The tutorial is strongly recommended for inexperienced users of TTP-Plan to get started, but it also provides some useful information for advanced users.
- Sections 3, 4 and 5 cover the interfaces for user interaction. Section 3 explains the graphical user interface, its menus, commands, editors and browsers. Section 4 introduces you to the programming interface; it gives a short introduction to the Python language and the possibilities of user interaction via scripts, as well as an overview of regular expressions that might be helpful. Section 5 finally presents the command line interface that allows you to start and use the tool without using the GUI. You also have the possibility to use a *batch mode* for executing several steps in series without intermediate interaction.
- Section 6 deals with the mechanisms of *schedule extension* and multiplexing, both of which provide a greater flexibility of a cluster schedule. Schedule extension enables you to add messages to an existing schedule, whereas multiplexing allows several nodes alternately use the same sending slot.

- Replication is used to improve fault tolerance, and replica-deterministic agreements (RDAs) are needed to ensure consistent output values. TTP-Plan supports several predefined agreements, which are described in Section 7.
- When a node fails temporarily, its subsystems should have a possibility to *reintegrate* in the system. The history state (*h-state*), explained in Section 8.1, provides system information to the reintegrating subsystems, thus enabling them to reintegrate in an optimal way. There are, however, even other reintegration strategies which are described in Section 8.2.
- Section 9 deals with *Remote Pin Voting (RPV)*, a mechanism that enables some nodes of a cluster to control another node and exclude it if it is considered faulty. This is an important safety enhancement.
- Section 10 describes the MEDL of the Monitoring Node which is slightly different from that of the cluster nodes.
- Section 11 presents the underlying object model of TTP-Plan and describes the object classes and associations. The subsections of section 12 contain the attributes of these objects with short explanations, mainly for reference purposes.
- Section 13 describes the use of the TTP M-Hub plugin.
- In the appendixes A and B you can find some details about the supported communication controllers and a list of Python scripts (*Goodies*) that facilitate or speed up some procedures for the user.
- Appendix C provides a glossary of frequently used terms and abbreviations.

2. Tutorial

TTP-Plan provides an on-line tutorial in the form of a step-by-step guide that takes the user through the complete process of the definition of a cluster and the generation of a cluster schedule. The step-by-step guide is selected by clicking on the ‘Guide’ tab of TTP-Plan’s main window (see Figure 3).

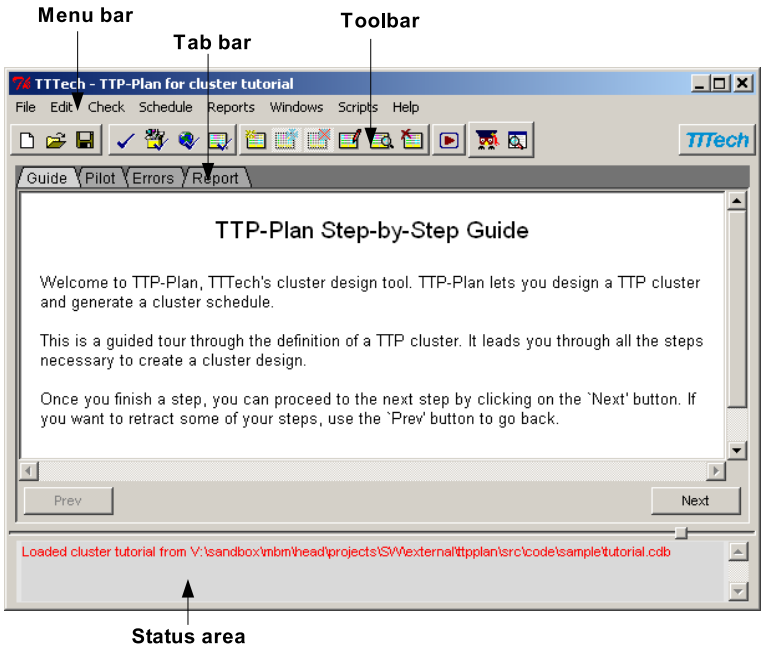


Figure 3: The Step-by-Step Guide of TTP-Plan

Cluster design requires information of the distributed application that will use the communication system, as well as knowledge about the principles of time-triggered distributed systems. This comprises basics, such as the round and the cluster cycle, but also more

advanced features like *redundancy* and *replication*, and TTP-specific topics such as mode changes, if these features are to be used in the design.

TTP-Plan is based on an object model that allows the user to design a cluster in a clear and consistent way; this model is described in Section 11. Understanding TTP-Plan's object model is an important step towards system design with TTP-Plan – the aim of this tutorial is to provide this kind of understanding by guiding you through the object model step by step. The step-by-step guide implemented in TTP-Plan can be used for the same purpose, providing immediate hands-on experience with the object model and TTP-Plan.

The guide is organized as a sequence of pages. Each page explains one design step and sports an action button which pops up the window required for this step (or directly carries out that step, if possible). You move between the individual steps by using the 'Next' and 'Prev' buttons which select the next and previous steps of the tutorial, respectively.

The tutorial introduces just one (recommended) sequence of steps to design a cluster with TTP-Plan. However, any sequence of object definitions is possible by using the Pilot window, which gives an overview of the whole object model and can be accessed by clicking on the 'Pilot' tab. But if the design is done as proposed by the step-by-step guide, all types of objects will be created in due time. It is really a question of taste whether, for example, messages are defined before or after subsystems. Depending on the data available, it may be advisable to first define the lot of messages and then assign them to the respective subsystems.

Note: If you intend to use TTP-Build or TTP-Simulate-Setup afterwards, you need to consider the type of communication layer you will be using (please refer to the TTP-Build manual for details). The basic settings in this tutorial are intended for FT-COM users, but the majority of them is compatible with both TD-COM, HW-COM and HS-COM. So whenever deviating settings are required for one or the other COM layer, this will be indicated in the text. Sample databases for use with each COM layer (`tutorial_*.cdb`) are provided in

`C:\TTP\<toolname>\<version>`.

They should be identical with the outcome of this tutorial, so you can also use them for comparison if you encounter problems.

To walk through this interactive on-line tutorial, please start TTP-Plan right now. You are greeted by the welcome page of the step-by-step guide. Please click the 'Next' button at the bottom of the window to continue.

2.1. Step 1: The Cluster – The Central Object of TTP-Plan

Cluster

The cluster object is the first and central object in a cluster design. It defines properties that need to be observed by all participants in cluster communication, such as the duration of a round, the type of communication controller, and the physical interface.

When TTP-Plan is started, no cluster object is present. Click on the 'Edit' button at the bottom of the Guide window: an input box pops up and asks you to enter a name for the cluster. Enter `tutorial` and confirm. Note that this name is the name of the clus-

ter object; it will usually also be used as the filename when saving the cluster database (`tutorial.cdb`).[2]

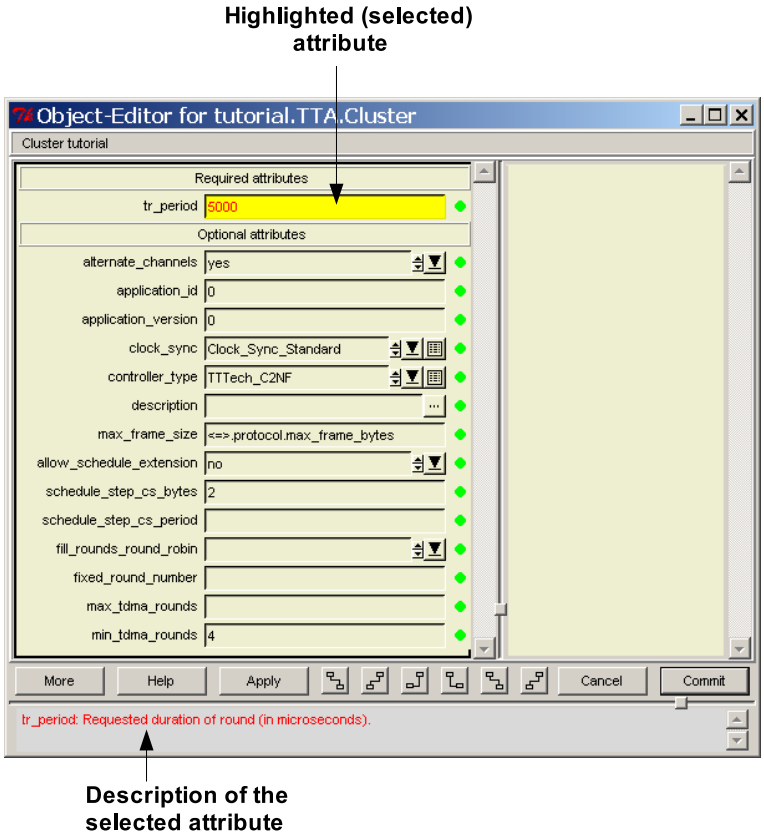


Figure 4: The cluster editor of TTP-Plan

After you have named the new cluster, TTP-Plan automatically creates a cluster object and opens an object editor which offers input fields for all attributes of this object (see Figure 4).[3] The attributes of the newly created cluster can now be filled with values – for most of them, there are reasonable defaults, but some must be supplied by the user. These *required* attributes are listed uppermost and are labeled as such. For the cluster object, there is only one required attribute: the `tr_period`. (For a complete list of attributes, please see Section 12.)

`tr_period`

[2] You are, however, free to choose a different filename. Changing the filename does not change the name of the cluster object.

[3] Usually, when opening an object editor, you would have to select one of the existing objects and then click the 'Edit' button to edit the specific object. Since there can only be a single cluster object present at a time, TTP-Plan automatically opens this editor for you.

Let's have a closer look at this attribute: this is the round duration, given in microseconds. The round is a basic notion of time-triggered communication; it defines a time interval which is then divided into distinct slots for the transmission activities of the nodes. Each node in a cluster can only transmit data once per round; therefore the round duration must not be longer than the shortest of all message periods in the cluster.[4] For the round duration there cannot be a reasonable default, since this depends entirely on your application. You may either enter the desired round duration here and have the scheduler check if such a duration is possible, or you can enter a ridiculously small value such as 1 microsecond to have the scheduler calculate the minimal `tr_period` for your application (and complain that your request cannot be satisfied after scheduling). round

Let us assume a round duration of 5 milliseconds for our application, so click into the `tr_period` input field and enter 5000. Note the information text in the status field at the bottom of this window. It gives a short description of the attribute; pressing the *F1* key displays an *attribute browser* which holds help information on this and the other attributes of the object you are currently editing.

The optional attribute `controller_type` accounts for the fact that different controller types can support different transmission speeds (or 'bit rates'). Select the controller type you are using from the drop-down list box. The bit rate will be entered in the next step (attribute `transmission_speed` of the `Web` object). `controller_type`

Note: if you are using a mixed (heterogeneous) cluster with different controller types, you need to set the optional attribute `controller_type` to `Controller_Set` (default value is `TTTech_C2NF`).

Note to HW-COM users: Please set the attribute `allocate_sender_status_at_start` to '8' to reserve the first 8 bytes of a frame or message box for the sender status. `allocate_sender_status_at_start`

All: We need not fill in any of the other optional attributes because there are reasonable defaults (visible in the respective input field). However, some of these defaults may not be appropriate for a specific design; for example, for the controller type to use for scheduling.

A good example for the use of optional attributes is the allowed number of rounds. There are two "scheduling strategies":

- "Power of two"-scheduler (default): here 1, 2, 4, 8, 16, ... rounds are allowed. However, one round will not be scheduled unless `min_tdma_rounds` is set to 1 (default is 2).
- "Fixed round number"-scheduler: here an arbitrary number of rounds may be specified by the user in the `fixed_round_number` attribute.

The maximum number of rounds depends on the controller type used.

One optional attribute that is always present is the `description`; this attribute can be used for documentation purposes, e.g., you can enter 'test cluster' as description of this `description`

[4] If the round were longer than the message period, the message would be sent twice in one round, which is not permitted.

cluster. Note the button next to the input field, containing three dots – this button allows to perform 'further editing' of the field contents. It pops up a larger editor window, allowing you to enter a long description if needed.

All other fields will be left unchanged for now; you can still click into any of them to find out what the status field has to tell about them.

When you are done, click the 'Commit' button. The information you supplied is now entered into the object model, and the object editor is replaced by an object selector. We will come across these two kinds of windows all the time; the *object* or *attribute editor* (often only called "Editor") allows you to view or manipulate the attributes of a specific object, whereas the *object selector* (also called "selector") is used to select an object (of a specific class) for editing. For most classes, many objects (a.k.a. 'instances') can exist, the cluster is an exception – there can only be one cluster in the database.

Click 'Close' to finish the cluster definition.

We are now back in the step-by-step guide. Please click 'Next' to proceed to the next step.

2.2. Step 2: The Web

A 'Web' object defines the attributes of a communication web. Most clusters will use only one web (named `<cluster-name>_web`), which is created automatically when you create the cluster object.[5] Have a look by clicking 'Edit' – there should be an entry named `tutorial_web` in the selector. Now double-click on it. You should specify values for the required attributes '`byte_order`' and '`transmission_speed`'.

`byte_order`

The `byte_order` attribute is necessary if nodes of different byte order have to participate in the same cluster. It specifies the 'default' endianness in this cluster, i.e., the byte order used for the transmission of data on the bus. For our example cluster, just choose one of the options present. (This information is used by code generating tools like TTP-Build and therefore not relevant for this tutorial.)

Note to HW-COM and HS-COM users: Please set `use_single_cluster_mode` to 'yes'. HW-COM only: set `byte_order` to '`little_endian`'.

`transmission_speed`

All: For the `transmission_speed`, a value of 5000 is a common default; this attribute is given in kbit/s, so 5000 amounts to 5 Mbit/s. The values that can actually be selected here depend on the type of communication controller that is used in this cluster (see the attribute `controller_type` of the `Cluster` object).

Entering a value that is not supported by the current controller type results in an error: the field containing the invalid value is highlighted, together with any other fields that are relevant for this error, and the condition causing the error is shown in the error display at the right side of the editor window. It must be corrected before the data can be 'committed'.

[5] If you design a multiweb cluster, you need to select the *active* web for downloading and/or viewing in the `Cluster` attribute `active_web`.

2.3. Step 3: The Host – Defining the Communication Participants

Host

When speaking of 'host' in the context of the TTP-Tools, what we mean is a node in the cluster, a computer, an ECU, a 'box' – an entity of computational and communicating capabilities. Let's define some hosts and see what they mean... Click the 'Edit' button of the guide.

Here you find a selector with one or more predefined "_mon" entries indicating the currently supported Monitoring Nodes, but no other host objects are present. If you select one of the Monitoring Nodes and click 'Edit', you will see that there are only optional attributes, so you do not need to enter anything here. To create the hosts we need, click 'New' – now the lower input field lights up, indicating that TTP-Plan expects your input here before a new host can be created. Enter `jack` and press the `<Enter>` key. – Why `<Enter>`? The `<Enter>` key will trigger the current 'default action', which is the last button clicked, and that is 'New'. Of course, you could still click on 'New', depending on your preference.

Host `jack` has now been created, let's enter `jim`, `joran` and `judy` in the same way; you will find that you have to delete the previous name to enter a new one. Also notice that the hosts are sorted alphabetically. The last host entered is selected with a yellow bar.

Now, let's assume a typo – we actually wanted to enter `jordan`, not `joran`. Select `joran` from the object list by clicking on it (you can also type `joran` in the upper input field, or select it in the drop-down list box of the upper input field), enter `jordan` in the lower input field,[6] and click 'Rename'. So much for that. Select `jim` and click 'Edit' – or just double-click `jim` – to see which attributes we have to supply. There are no required attributes for the hosts, but we can supply a number of `optional` attributes to influence the schedule generation (e.g., `min_frame_size` or `max_frame_size`).

Actually, why don't we make life a little easier for us and the scheduler? As you will learn in Section 11.1, TTP-Plan uses *frames* to transmit data between the nodes. There are different types of frames: I-frames, N-frames and X-frames (see the glossary in Appendix C for detailed descriptions). Only X-frames contain both initialization information *and* application data and can thus be transmitted in every slot in every round. Although this adds a certain transmission overhead, it provides the highest flexibility, so we will tell our hosts to make use of X-frames.

Goodies

`use_x_frames`

For the job at hand, a Python script (see Section 4) is already installed – you can run it by selecting 'Scripts' from the menu, select the 'Goodies' category and run the script called `use_x_frames`. That's all. Now open the editor for one of the hosts and check out the attribute `use_x_frames`: it should be set to "yes". That's exactly what the script did. The default for this attribute is "no", but the script sets it to "yes" for all hosts, so you don't have to do it manually and one by one.

[6] Especially for longer names, copy and paste are nice to use for this purpose; TTP-Plan supports the standard hot-keys Ctrl-X, Ctrl-C, and Ctrl-V for cut, copy, and paste, respectively.

Note to HW-COM and HS-COM users: As these COM layers only support a single cluster mode, we need to prevent our hosts from attempting a cluster mode change. Please set the attribute `allow_mode_change` to 'no' for all hosts. `allow_mode_change`

All: Most objects have more attributes than are displayed here, but only those displayed can be changed by the user. By clicking the 'More' button, other attribute groups become visible. Just try this now...

Below all the `optional` attributes, there are now several other sets of attributes visible – they often have cryptic names ('computed', 'internal', 'schedule', 'cached') and are displayed in a different style, with grayed-out values instead of input fields. These are *view-only* attributes of the host object; they are set or calculated by TTP-Plan automatically and cannot be changed directly by the user,[7] although they may be listed, for example, in automatically generated bug reports. *Note:* you can view the descriptions of all view-only attributes in the same way as for editable ones. Just click on the displayed value of an attribute to see its description in the status area.

Select 'Close' now to leave the host object selector altogether. We have now defined a cluster and four hosts.[8]

What happened in the background

Behind the curtains, TTP-Plan has already done something for us. Let's take a look at the 'Pilot' tab:

- `Host_in_Cluster`: `Host_in_Cluster`

  Move the mouse to the link `Host_in_Cluster` between the 'Cluster' and the 'Host' object so that the line and the word 'in' become highlighted red, then double-click on the link, and you will find four entries in the link selector that opens up. This means that the four hosts we just created are already linked to the cluster. TTP-Plan can do this automatically because there can only be one cluster at a time, and all hosts must belong to it.

  There is one required attribute for this link – the `serial_number`. This is a unique number that identifies this host in the cluster.[9] Enter `1` as serial number for the link between `tutorial` and `jack` and 'Commit' it. Then 'Edit' the others, giving them unique serial numbers like 2, 3, and 4. (For the tutorial, it does not matter which number is assigned to which host, only that each host gets a unique number.) 'Commit' each of the links, then click 'Close' to exit the selector. `serial_number`

  Note: the Monitoring Nodes are not linked automatically, because this is only required if you want to have so called "active monitor hosts". In this case you have to create the link to the cluster manually.

- `Host_in_Web`: `Host_in_Web`

[7] Some of these attributes can be changed indirectly by setting optional attributes or manually editing the schedule.

[8] Four is the minimum number of hosts for a fault-tolerant cluster.

[9] This number is used by TTP-Load for download to this node.

All hosts in the cluster also need to be linked to the same web the cluster uses. If you now click on the link `Host_in_Web`, you will see the corresponding links for all hosts in the selector, connecting them to the web object `tutorial_web`. You do not need to edit any attributes here at the moment.

- `Slot`: `Slot`
  TTP-Plan has also created '`Slot`' objects for your new hosts and linked them to the 'Host' objects. If you double-click on 'Slot' in the 'Pilot' view you will find the automatically created objects like `jim_slot` for 'Host' `jim`. There are some optional attributes for the slots, but you don't need to edit them. Return to the pilot and double-click on the link `Host_uses_Slot` between the 'Host' and the 'Slot' object now; you will see the appropriate entries, for example `jim` and `jim_slot`. `Host_uses_Slot`
  While the automatic generation of a separate 'Slot' and the respective link for a new 'Host' object is most convenient, you will have to change links and delete the unused 'Slot' objects if you want to use *multiplexing*. When using multiplexing, two or more hosts use the same sending slot in different rounds (see the description in Section 6.3 for details). The two attributes `mux_period` and `mux_round` are needed for this purpose.

To proceed, please switch back to the 'Guide' window and click 'Next'.

2.4. Step 4: Host Runs Subsystem in Cluster – The First Link

We now have a cluster object and some host objects. Next we are going to define which tasks these hosts execute. Please click 'Edit' to open the selector window, if you have not already done so.

You now see a *link selector*, and true to the "hardest-first" principle we start with a ternary link: three different objects are "linked" together by one link. You can see the three object classes on top of the lists: TTP-Plan now expects you to specify a *host*, a *subsystem*, and a *cluster*, and will create a link between these three. A *link* describes a relation between the linked objects, whereas an *association* describes the general relation between the corresponding *classes*. The "function" of the relation is directly visible from the name of the association – in this case, 'Host runs Subsystem in Cluster'.

We already have a cluster and hosts, but no subsystems. What is a subsystem, anyway? You can think of it as a set of tasks that take some input and produce some output. Several subsystems may be executed – independently of each other – on one host (that is, if it has enough processing power for this), and one subsystem may be executed simultaneously on several hosts in a cluster – we call this a replicated subsystem. `Subsystem`

In other words, the term *subsystem* is used to describe a specific functionality performed in the distributed system. In the context of TTP-Plan, a subsystem is solely defined by a name and the messages it sends into the system – in the context of a system design, it may be something like 'the I/O controlling the engine' or 'an anti-lock-brake control algorithm'. Many designs simply associate host and subsystem one-to-one; you could do this here as well, defining `jim_subsystem` as the only and full functionality performed by host `jim`, but the design allows much more flexible definitions.

Ultimately, this concept allows to easily define (and change) redundancy concepts on the subsystem level – replicating a sensor subsystem does not necessarily mean that all functionality of the sensor ECU is replicated (e.g., speed sensing is replicated, but temperature sensing is not). Distinguishing between the two can hence bring a lot of flexibility into the design.

For our example cluster, we will define three subsystems, and call them ‘ear’, ‘eye’, and ‘nose’. Then we have to tell TTP-Plan which host executes which subsystem. Let’s define them as follows:

- host `jack` will do nothing,
- host `jim` will execute ‘ear’,
- host `jordan` will execute ‘nose’ and ‘eye’, and
- host `judy` will execute ‘eye’.

This means that there are no two equal hosts, but the functionality represented by ‘eye’ exists twice (redundantly) in the system; ‘ear’ and ‘nose’ just exist once.

Let’s start with ‘jack’. Select host ‘jack’, either by typing `jack` in the input field in the ‘Host’ column close to the ‘New’ button, by using the small up/down arrows (spinner buttons) next to this field, or by opening the drop-down list and clicking on ‘jack’. Note that you must use the lower field because we want to create a new link; the upper input field (right above the ‘View’ and ‘Delete’ buttons) is used for selecting already existing links. So please keep in mind: when creating a new link, the names for the objects to be linked must be entered or selected in the lower input fields, above the ‘Select filter’ buttons.

Next, we will implicitly create a subsystem. You can check that no subsystems are currently defined by opening the drop-down list box in the ‘Subsystem’ column – no entries. Now click into the lower input field in the subsystem column, enter `nothing`, then press the tab key once to move into the next input field (or click directly into the cluster input field).

You will notice that you don’t really have a choice; the only cluster you can choose is `tutorial` because you cannot create a new cluster here. Select `tutorial` and press ‘Enter’ (or click the ‘New’ button). The link is created and listed as the first (and only) entry in the link selector. If you open any of the drop-down list boxes in the subsystem column now, you will find `nothing` listed there – before, there was no entry. We have just implicitly created a subsystem called `nothing`. This is not just for fun – what we are actually doing here on a system level is defining a node that will not send any application data, but still have a sending slot. This can be useful if a schedule is packed full with application data and does not have any nodes that send I-frames[10] – a node like ‘jack’ will send *only* I-frames, performing as a so called ‘I-frame server’.

Let us define the other subsystems and links; select ‘jim’ as the target host, enter `ear` instead of `nothing` in the subsystem input field, and make a new link.

[10] Please see the “Time-Triggered Protocol TTP/C High-Level Specification Document, Protocol Version 1.1”, document number D-032-S-10-028, for details on why some nodes must send I-frames.

Next, select ‘jordan’, enter `eye`, and make a new link. Do the same for `jordan`’s `nose` – no, please make a mistake and create a link to subsystem `noose` instead. This means: a subsystem `noose` is automatically created, but we don’t really want it!

TTP-Plan will assume that if you delete or rename the link again, you didn’t really mean to create this subsystem, and will delete it automatically. So just select the erroneous link, enter the correct name `nose` for the subsystem (in the lower input box, of course!) and click on ‘Rename’. When you now open the drop-down list of subsystems, you won’t find `noose` in there anymore...

When defining the links for ‘judy’, the subsystem name already exists in the drop-down list box for `Subsystem`, so you can either enter it manually or just select it using the mouse. Be careful to click on the ‘New’ button and NOT use the ‘Enter’ key when you define the link for ‘judy’ – the default action is now set to ‘Rename’ because we did a rename operation before, and therefore the ‘Enter’ key performs a renaming operation until you click on a different button.

When you have created all links – there should be five of them, two for ‘jordan’ and one for each of the other hosts –, close the selector and proceed to the next step of the guide.

2.5. Step 5: Subsystem Sends Message

When you choose ‘Edit’ now, another link selector window comes up. This link is simpler than the last, because it only connects two objects: subsystems and messages. Here we define which pieces of information are generated by which subsystem.

We have already defined subsystems. Now we will tell TTP-Plan that

- `ear` yields messages `loudness` and `sounds_nice`
- `eye` exports `brightness` and `color`
- `nose` transmits `sweet` and `yuck`

Subsystem `nothing` sends exactly that, so we will not specify any messages coming from that subsystem.

(*Note to CAN users:* The notion of ‘message’ as we use it here translates to ‘signal’ in CAN language. CAN’s ‘messages’, in turn, correspond to TTP ‘frames’. We do not use the term ‘signal’ in TTP context because it indicates an *event*, whereas TTP messages typically contain *state* information.[11])

We do not yet specify the data types of any message. Instead, we just create links, implicitly creating the messages: Select subsystem `ear` (in the lower input field, as always) and enter `loudness`, creating the link and the message with ‘New’. Just for fun, enter `sounds nice` (with a blank in between) instead of `loudness` now and try to create it. You will find out that TTP-Plan does not accept this name (because it would not be valid in the C programming language). Correct this to `sounds_nice` and create the link again. *Some hints for keyboard aficionados:* the tabulator key moves the focus between the input fields, the fastest way to clear an input field is the key combination `<Home> <Ctrl-k>`,

[11] See Section 11 for a description of state messages.

the fastest way to enter a new value in a field when the cursor is already at the end of the field (e.g., from previous typing) is <Shift-Home> (this highlights the whole field contents) and typing the new value.

Input fields have a 'history' that allows to quickly access the values previously typed in each field: Pressing <Alt-Up> selects the next older value, <Alt-q> shows a menu with all values entered for this field.

Now link subsystem eye to the new messages brightness and color – this should be no problem now. Same for subsystem nose: it will transmit the messages sweet and yuck.

There are now six links in this association. Imagine we had six hundred instead – how would we find just the ones belonging to subsystem nose? In TTP-Plan, you can set a lock on any of the object lists to filter the displayed links. Select subsystem nose in the upper(!) input box by clicking on one of the two links containing this subsystem, click on the blue 'a-z' 'action button' and select 'toggle lock'. You will see a small lock sign instead of the 'a-z' now, and the link list only displays two links now – the two links we defined for subsystem nose. Release the lock in the same way as you set it to see all six links again.[12]

By the way, if you lock to a name that is not connected to any link of the current association, e.g., if you lock subsystem to ear before defining the links for it, the list of links will be empty.

The links are currently sorted by the subsystem name. If you want to sort by message name instead, you can use the button in the 'Message' column, which currently displays three dots.

Click it, select 'sorted by', and the sort key symbol 'a-z' will move from the subsystem action button to the message action button. The list of links is now sorted alphabetically by message names.

Note that the 'locked' symbol overrides the 'sort by' symbol, because if a column is locked, it only contains one value, so sorting is not relevant.

We are finished with the link TTA.Subsystem_sends_Message, so close the selector. Proceed to the next step by clicking 'Next'...

2.6. Step 6: Message Types

In the previous step, we defined messages without saying what kind of messages they really were. But for scheduling data transmission, we need to know 'how many bytes/bits is this message long', i.e., the size of the message. Since there may be lots of messages, but probably many of them of the same type and size, we economize by defining separate objects for such *message type* information, and link them to the messages. These objects are called TTA.Msg_Type.

TTA.Msg_Type

In a typical design, we will probably encounter byte values (8 bits long), there may be unsigned integer values (say, 16 bit) or signed long (32 bit), but also application

[12] There is another way of restricting the objects displayed in the combo box and the target drop-down lists, which offers a different kind of filtering useful for completing or correcting input. See Section 3.6 for a description of the 'filter' mechanism.

specific types like `i/o sample` (10 bit) or flags of type `boolean` (1 bit). You will already see some predefined types in the selector now, like `ubyteX` and `sbyteX`. [13] All these types are so called "primitive" message types (`TTA.Msg_Type_P`).

There are other, "structured" types as well, which are slightly more complex to define. They are described in detail in Section 11.1.[14]

FT-COM, TD-COM and HW-COM

Since we don't have any space to waste, we will transmit 10 bit sized messages as 10 bits and not use 16 bits for them, if at all possible! The space a message of each type requires in a frame can be stated in bytes and/or bits and therefore may differ from what the C function `sizeof` would return.

So let's create some `Msg_Type_P`. We'll assume our application uses

```
typedef unsigned short uword;      /*16 bit values*/
typedef unsigned char   bool;      /*has just 1 significant bit*/
typedef char       char_6bit;      /*1 byte in memory,
                                    but only the lower 6 bits
                                    are significant information*/

brightness, color, loudness:    uword
sounds_nice:                     bool
sweet, yuck:                 char_6bit
```

and we will tell TTP-Plan about these types.

Enter the object selector for `Msg_Type_P` with the 'Edit' button of the guide and have a look... As you can see, some of the most common types are predefined and already present. Now create a `Msg_Type_P` named `uword` (by typing into the lower input field and clicking the 'New' button...). Let us see what we have got now: Select `uword` and click 'Edit'. The first and most important attribute of a `Msg_Type_P` is the `length`. You can enter the length in bytes, like '4' (means four bytes), or in bits, like ':12' (twelve bits). This can even be mixed to enter something like '1:2' (one byte, 2 bits, gives 10 bits in all). The size of a `Msg_Type_P` can range from 1 bit to as many bytes as fit into a single frame. For our new `uword` type we use '2'.

`length`

The second required attribute is the `type_cat`, the category of type in a programming environment. This is important for later processing of data of this type, e.g., by code generating tools. For simple variables like we have here, the type category is either `INT` (for variables which can become negative) or `UINT` (for variables which can only be positive). The `uword` is always positive (u meaning 'unsigned' meaning always positive) and therefore must be of type `UINT`.

`type_cat`

[13] The naming convention for these is as follows: "s" stands for "signed", "u" for "unsigned", and "X" is the length of the message in bytes.

[14] You can see all message types in the 'Pilot', when you click on `TTA.Msg_Type`.

Finally, enter `unsigned short` in the (optional) `typedef` attribute field[15] and ignore the other input fields. `typedef`

'Commit' your settings and create a `bool` type as well, with a length of ':1' and `UINT` as `type_cat`. After committing this, create another new type, `char_6bit`, with a length of ':6' and a `type_cat` of `INT` (because variables of this type can become negative, say).

All three types successfully committed, i.e., without any error messages? Fine, then close the selector and take another step in the guide. We are already halfway through!

HS-COM

As the HS-COM is optimized for fast CPU access, its range of "transmittable" data chunks is limited to multiples of 64 or 128 bits. Essentially, we could also just define empty message boxes of suitable size, specify the way they are accessed by the HS-COM in TTP-Build, and leave it to the application (designer) to fill them appropriately. However, as this is a training session, you might as well get to know a bit about message definition as well.

We'll assume our application uses

- A 2-byte `Msg_Type_P` named `uword` for smaller messages (we'll explain later how they can be used with the HS-COM).
- A 64-bit array (`Msg_Type_A`) of `ubyte1` elements, named `array_64`, for simple state messages.
- A 128-bit array (`Msg_Type_A`) of `ubyte1` elements, named `array_128`, for event messages.

Later on, we will assign these message types to our messages as follows:

```
loudness:                          uword
sounds_nice, brightness, color: array_64
sweet, yuck:                    array_128
```

Enter the object selector for `Msg_Type_P` with the 'Edit' button of the guide and have a look... As you can see, some of the most common types are predefined and already present. Now create a `Msg_Type_P` named `uword` (by typing into the lower input field and clicking the 'New' button...). Let us see what we have got now: Select `uword` and click 'Edit'. The first and most important attribute of a `Msg_Type_P` is the `length`. You can enter the length in bytes, like '4' (means four bytes), or in bits, like ':12' (twelve bits). This can even be mixed to enter something like '1:2' (one byte, 2 bits, gives 10 bits in all). The size of a `Msg_Type_P` can range from 1 bit to as many bytes as fit into a single frame. For our new `uword` type we use '2'. `length`

The second required attribute is the `type_cat`, the category of type in a programming `type_cat`

[15] The `typedef` attribute is not required in TTP-Plan; however, entering valid C typedefs serves as input information for code generation and on-line monitoring later on.

environment. This is important for later processing of data of this type, e.g., by code generating tools. For simple variables like we have here, the type category is either `INT` (for variables which can become negative) or `UINT` (for variables which can only be positive). The `uword` is always positive (u meaning 'unsigned' meaning always positive) and therefore must be of type `UINT`.

Finally, enter `unsigned short` in the (optional) `typedef` attribute field[16] and ignore the other input fields. `typedef`

'Commit' your settings for `uword` and close the selector afterwards.

Now for the "big guys": to define your arrays, open the selector for `Msg_Type_A` from the 'Pilot' page and create `array_64` and `array_128` the same way you just created `uword`. There are no required attributes to set here, but there is something else to be done instead, namely defining the elements of the arrays. This is done by means of `TTA.Msg_Type_A_uses_Msg_Type` links. Go to the pilot again and double-click on this association. In the link selector, create a link between `array_64` and `ubyte1`, and double-click on it. To define the length of an array, you have to specify its `bounds` (in bytes). Since we want our array to be 64 bits long, enter "8" here and commit this.

Now create a link between `array_128` and `ubyte1` and set `bounds` to "16". Commit, and you are done. Next we will see where these message types appear in the messages.

2.7. Step 7: Completing the Message Definitions

Let's give a moment's thought to the messages exchanged in our cluster. They have already been created (implicitly, by defining the links `TTA.Subsystem_sends_Message`), but not yet been completely defined. In other words, TTP-Plan knows the names of the messages, but nothing else. Is there anything else we need to supply for message objects? Enter the selector for the class `TTA.Message` (by clicking 'Edit' in the guide, as you have taken the 'next' step already) and take a look at the list of already defined messages. Edit `brightness` by double-clicking it. `Message`

There are two required attributes, the `init_value` and the `msg_type`. Let's take a look at the `init_value` first: This value is needed during cluster startup; see "`TTA.Message`" in Section 11.1 for some details. For our example you can set this attribute to '0' (zero) for all messages. `init_value`

If you want a different `init_value` for all six defined messages, you could edit one after the other, change the value, commit – sounds like a job you don't want to do yourself if there are a lot of messages, right?

If you don't want to waste energy, you can (and should) use a *script*. Scripting is a powerful way to extend TTP-Plan's capabilities, to customize it, and especially to automate repetivite tasks like this one. The script language is called Python – a widely available and hugely popular free interpreted language (check out http://www.python.org). Scripting

As for setting `use_x_frames` in Step 2.3, a 'Goodie' script for setting the `init_value` is Goodies

[16] The `typedef` attribute is not required in TTP-Plan; however, entering valid C typedefs serves as input information for code generation and on-line monitoring later on.

also available. Select ‘Goodies’ from the ‘Scripts’ menu and run the script `set_init_values`. Enter the desired new `init_value` (e.g., 10). In the status area at the bottom of TTP-Plan’s main window you can see a message saying how many messages have been updated with that value. *Note:* this script just updates messages that don’t yet have an `init_value`, so if you run it again the number of messages updated is zero. You could now run another script called `change_init_values` to change all messages which currently have ‘10’ as `init_value` to a different value, and so on.

Scripts are a highly useful feature of TTP-Plan, also if data has to be exported or imported!

Now turn your attention to the second required attribute `msg_type`: It is needed to provide information about size and type of a message to the scheduler. In the previous step we defined three such message types, now we are going to use them according to the scheme outlined there: `msg_type`

FT-COM, TD-COM and HW-COM

Remember, we assumed that our application uses the following types:

```
uword:     brightness, color, loudness
bool:                        sounds_nice
char_6bit:                   sweet, yuck
```

HS-COM

Remember, we assumed that our application uses the following types:

```
uword:                             loudness
array_64: sounds_nice, brightness, color
array_128:                    sweet, yuck
```

All

So double-click on each message in turn, enter the appropriate `msg_type` and click ‘Commit’. While doing this, you can also verify that the `init_value` is indeed ‘0’ for all messages. Just for fun, let’s make a mistake here: “mislink” `sweet` to `bool` and commit this. When you have edited all messages, close the object selector, but before taking the next step in the guide, let’s take a look at what happened in the background: switch to the ‘Pilot’ tab and double-click on the association `TTA.Message_uses_Msg_Type`. Probably you can already guess what we will see here … Exactly, the `msg_type` attributes were transformed into links. All messages are there, linked to their message types as specified.

Now what about that mistake we made? If an unwanted link is created (like `sweet` linked to `bool`), there are two ways of handling this in the link editor (of course, you can also fix it by returning to the `TTA.Message` editor and changing the `msg_type` there):

- Delete the unwanted link and create the correct one (straightforward, but more work)
- Rename the unwanted link in a way to get the right link; in this example, click on the link, change `bool` to `char_6bit` (`array_128` for HS-COM) in the lower input field for `TTA.Msg_Type` and click on 'Rename' (recommended)

TTP-Plan will not let you link a message to more than one `TTA.Msg_Type` since that does not make sense, so you cannot first create the correct link and then delete the unwanted one.

Before leaving the link selector, let's go for another experiment: Please delete the link between `sweet` and `char_6bit`/`array_128` by clicking on it and then clicking the 'Delete' button. Done? Great, then close the selector and take another step...

2.8. Step 8: Message in Message Box (HW-COM and HS-COM only!)

As both COM layers can only handle data in chunks of a certain size, we need to find a way to "wrap" our messages in packets of appropriate size. This is done by using *message boxes (`TTA.Msg_Box`)*.

Click the 'Edit' button to open the link selector for `TTA.Message_in_Msg_Box`. Here you can create the needed message boxes implicitly, the same way you created the subsystems together with the `TTA.Host_runs_Subsystem_in_Cluster` links. Assume we want to put `ear`'s two messages `loudness` and `sounds_nice` into two separate message boxes named `ear_box_1` and `ear_box_2`. Select `loudness` in the lower left input field and type `ear_box_1` in the lower right one. Then click 'New'. Select `sounds_nice` on the left side, change `ear_box_1` on the right to `ear_box_2` and press 'Enter'. See? Two links already in place!

For the other two subsystems, we will make it easy for us and put each subsystem's two messages into one message box: Put `brightness` and `color` into `eye_box`, and `sweet` and `yuck` into `nose_box`. The subsystem `nothing` doesn't need a message box because it doesn't send any messages. When you are done, double-click on one of the links to check for attributes – nothing required here, so you can leave it at that and close the selector.

HW-COM

What's next? You remember that we need 4-byte chunks, so let's see if we can provide some kind of size information – double-click on `Msg_Box` in the 'Pilot' tab and open the editor for `ear_box_1`. The required attribute `data_size` is just what we need, so please enter '4' here.

`data_size`

Before you close the editor, however, take the opportunity to fix another thing we need along the way: Since message boxes are in fact messages themselves, we also need to specify their sending subsystems as well. Conveniently, we can do this right here, instead of changing to the `TTA.Subsystem_sends_Message` selector. The last optional attribute `subsystem` is all we need. Select `ear`, then click 'Commit'. Continue with `ear_box_2`,

eye_box and nose_box in the same way, commit them and then close the selector. Now we are ready to continue with the Guide...

HS-COM

What's next? You remember that the HS-COM only accepts multiples of 64 or 128 bits, so let's see if we can provide some kind of size information – double-click on Msg_Box in the Pilot and open the editor for ear_box_1. The required attribute data_size is just what we need, so please enter '8' here. data_size

Before you close the editor, however, take the opportunity to fix another thing we need along the way: Since message boxes are in fact messages themselves, we also need to specify their sending subsystems as well. Conveniently, we can do this right here, instead of changing to the TTA.Subsystem_sends_Message selector. The last optional attribute subsystem is all we need. Select ear, then click 'Commit'.

Continue with ear_box_2, eye_box and nose_box in the same way, assigning them to their respective subsystems and setting their data_size as follows:

- ear_box_2: 16
- eye_box: 16
- nose_box: 32

When you have committed all of them, close the selector. Now we are ready to continue with the Guide...

2.9. Step 9: Linking Messages to the Cluster

Now comes the last big step to a complete cluster design: we must tell TTP-Plan how to use the messages for transmission. This seems superfluous, since we already defined subsystems and the messages they transmit, and we already defined the hosts that execute these subsystems. However, we still have to supply aditional information about the messages in our cluster, such as whether they should be transmitted redundantly (i.e., on both channels) or not, using the attribute redundancy_degree.

FT-COM and TD-COM

Open the link selector and create a link between the cluster tutorial and the message 'brightness'. After creating the link, click 'Edit' to look at its attributes.

There is one required attribute, the d_period, which stands for 'design period'. This is the (maximum) transmission period of the message in this cluster, and the scheduler will strive to have this message transmitted (at least) once every d_period. If this is not possible, we will get a scheduler error. It is OK for the scheduler if the message is transmitted more frequently than specified; the scheduler will tell us in the a_period ('actual period') attribute of the message what the actual update period is after the schedule is done.[17] d_period

[17] Example: Stating a d_period of 5 ms with a round of 4 ms will result in an update period (a_period) of 4 ms. If the message is a real state message, a higher update rate cannot make the 'quality' of the data

For ‘brightness’, let us assume a maximum transmission period of 10 milliseconds, so enter ‘10000’ here. *Note:* these are microseconds! Many designs state millisecond units for update rates, so please be careful...

min_round
max_round

The optional attributes are interesting as well: with `min_round` and `max_round` we can specify a portion of the cluster cycle during which the message must be transmitted first. This defaults to ‘1’ (the first round) and `TTA.Cluster.max_tdma_rounds`, which stands for the highest round number in the (resulting) schedule.[18] These attributes can be used to enforce phase sensitivity between messages – say, if a cluster design explicitly states that message `sensor_data` is transmitted in the first round of the cycle, and the resulting `valve_command` is transmitted in the third round, these dependencies can be entered here.[19]

redundancy_degree

Let’s force message `brightness` into the second round, just to see what happens. Enter ‘2’ in both `min_round` and `max_round` fields. The `redundancy_degree` is another important attribute: it defines the number of channels to be used for the transmission of the message. If set to ‘1’ (which is the default), we are satisfied with transmission on only one channel, which reduces the bandwidth requirements of the system, but does not satisfy fault tolerance requirements against failure of a communication channel. Let us assume that `brightness` is an important message for the system, so we set the `redundancy_degree` to 2.

Hint: If a design says to transmit all messages redundantly, there is another Python script (`set_all_messages_for_redundant_transmission`) in the ‘Goodies’ directory. It is very simple (view the source with any text editor!) and is run once after all messages have been defined, but before a schedule is made.

sampling_factor
sampling_phase

The attributes `sampling_factor` and `sampling_phase` are important for task scheduling. The `sampling_factor` gives the number of transmissions per calculation of this message, the `sampling_phase` specifies the phase of the sending task.

Typically, a task is executed exactly at the same rate as the messages it sends, and vice versa. That means, all messages sent by the same task must have the same period (i.e., how often they are generated).[20] If a message is sent more often than the task is executed (that means, the `sampling_factor` is greater than one, i.e., “oversampling”), the same message copy will get transmitted several times. See Section12.2 for details.

For our example you can keep the defaults here. ‘Commit’ your changes now and move on to the next message:

For the use of the message ‘color’ in our cluster, we assume the same properties as for ‘brightness’. We therefore want to copy the link we just created: select ‘color’ in the target field and click ‘Copy’. That’s it. To see if the link was properly copied, edit it (double-clicking works here as well). Seems OK: `d_period` is 10000, `min_round` and `max_round` are 2, `redundancy_degree` is 2. Cancel or commit, we don’t need to save anything as ‘Copy’ already did all the work for us.

---

worse. However, for control loops that require an update rate of that message of ‘exactly’ xxx ms instead of ‘maximum’, the message update period must be a power-of-two multiple of the round duration.

[18] See also Step 1, “Cluster” for more information about allowed numbers of rounds.

[19] Try to avoid such phase sensitivities, if possible.

[20] A task can, however, *receive* messages with different periods.

For the other messages, we want them to be transmitted every round. To avoid having to correct our input in case the round duration ever changes, we can use a symbolic value instead of a literal number: `Cluster.tr_period` – this is the value of the attribute `tr_period` of the cluster object. Remember, this was the first attribute we entered when defining the cluster object. *Note:* such symbolic expressions only work if the attribute they refer to has been specified. You cannot set the `d_period` to `Cluster.tr_period` before you have specified a value for `tr_period`.

So create a link between `tutorial` and message `loudness`, edit this link and for the `d_period` enter `Cluster.tr_period`. We don't have any objections to the default values in the optional attributes fields, so just commit this link and copy it for the other three messages (`sounds_nice`, `sweet`, and `yuck`). *Note:* if you place the cursor in the lower input box and just press the up/down arrow keys on the keyboard, you can quickly scroll through the messages. You can then either click on 'Copy' or – if that is the default action already – just press the 'Enter' key. Works really fast!

Now, we do want message `sounds_nice` to get redundant transmission, so edit the link `tutorial<->sounds_nice` and set the `redundancy_degree` to 2. Commit this change and close the selector, we should be done with the data input.

HW-COM and HS-COM

For the HW-COM and HS-COM layer, the situation is a bit different because all messages are now contained in message boxes. However, this is actually an advantage, as we don't need to define `TTA.Cluster_uses_Message` links for all messages. It's sufficient to link the boxes to the cluster, the messages are linked automatically. This is the most common use case for message boxes, to reduce the number of transmissions. Of course, you are free to read through the above definitions as well, there are some interesting things about the optional attributes in there.

To go straight ahead, please click 'Edit' and create links between the cluster `tutorial` and the four message boxes, then open the editor for the first link (between `tutorial` and `ear_box_1`): There is one required attribute, the `d_period`, defining the (maximum) transmission period of the message in this cluster (see above for a more detailed description). Enter `Cluster.tr_period` here to specify the same value as for the attribute `tr_period` of the `Cluster` object. Do the same for the `ear_box_2` and `nose_box` links, but set the attribute to '10000' for `eye_box`. The other attributes can remain untouched for all four links.

Regardless of the COM layer you are preparing for, this was the last step in the cluster design. So now we will check if we made any mistakes... Close the link selector and take another step in the Guide by clicking 'Next'!

2.10. Step 10: Checking the Design for Errors

Checking for errors is an easy way to find out if we have forgotten anything, like linking a message to the cluster or a subsystem. TTP-Plan will also complain if some values

we entered are out of specification. The errors are listed in the error browser and can be corrected by clicking on the culprit and editing the corresponding attributes.

Try checking for errors now by clicking the ‘Check’ button!

And indeed, TTP-Plan finds one error. (If you followed this tutorial precisely, there should be exactly one of them...) The error browser lists the encountered errors in different groups. You see that there are no errors in the ‘object-internal invariants’; [21] the error detected is an ‘object-integrity’ error, i.e., some object definition is incomplete.

This means: our object definitions are complete and correct, but TTP-Plan still detected something amiss. What could that be? Right-click on the black triangle left of the ‘errors’ message to expand two levels of the ‘error message tree’ – left-clicking opens just one level, which is not always enough to see what the problem is, so you should get used to right-clicking the triangles in the error browser.[22]

So what is the problem?

Maliciously, you were instructed to delete the link between `sweet` and its message type. Looking back, this made no sense – we were just simulating a forgotten input. And now TTP-Plan is complaining that the attribute `msg_type` is not defined for `sweet`.

We did this just to show that TTP-Plan checks before making a schedule (in fact, it checks a lot of things all the time, before and after scheduling) – let’s correct the problem: click on the underlined `sweet` to get to the object editor, and set `msg_type` to `char_6bit` (`array_128` for HS-COM) as originally planned. Then close the editor again. You may notice that the error tree is not immediately updated – checking for errors may take some time, so you can make any number of corrections, then run the checks again, and only then you will see if you have removed all errors. But, of course, you can run the checks after each correction if you desire.

Click on the blue check mark in the icon bar, or select ‘Everything’ from the ‘Check’ menu (this is the same action the guide did a minute ago); the checks are run again, and now all errors should be gone. We are ready to make a schedule. Click on the ‘Guide’ tab to continue.

2.11. Step 11: Running the Automatic Scheduler

The next step of the guide lets us run the automatic scheduler of TTP-Plan. Here, the cluster design we entered is taken as input from which a cluster schedule is generated as output, consisting of the objects shown in *white* in the Pilot window, and the appropriate links.

When these objects and links have been created, another error check is run, this time testing if all the ‘schedule invariants’ are satisfied by the new schedule. There are many reasons why a correct input design can lead to erroneous schedules!

[21] An invariant is a logical condition which must be satisfied at all times. An object-internal invariant only protects the object itself against faulty inputs, it doesn’t care about completeness or correct interactions with other objects.

[22] You also can set the focus on any item and press `Ctrl+Insert` to expand all levels below it.

Let us try if our small design can be successfully scheduled: click ‘Schedule’ in the guide. Some progress gauges fly by, then a window

‘Summary of message schedule of cluster tutorial’

pops up, showing the ‘schedule summary’.[23] Here you can check out basic properties of the generated communication schedule, like the bandwidth usage and remaining bandwidth (‘stretch’) on the bus, as well as some information about the timing (e.g., “N-frames per second”, “messages per second”, etc.).

If there were any errors in the schedule, the error browser would have come up and told us about this, but in this example the schedule was made without errors. (You can still click on the ‘Errors’ tab to see that it now says ‘No errors in schedule invariants’.)

To find out what TTP-Plan did, let us first check if any new objects have indeed been created! Click on the ‘Pilot’ tab and double-click the Slot class box. Here you can see the four objects of class ‘Slot’. Now let’s have a look at jack_slot. To do so, select jack_slot and click the ‘Edit’ button. Slot

Click now on the ‘More’ button and look at the ‘Schedule attributes’. All fields are now read-only, but some of the generated attribute values may be interesting for the host programmer; for example, you can see the duration of the slot (in macroticks), or the durations of transmission_time, idle_time and some more in the slot. Here is a nice shortcut to remember: right-click on a ‘Cancel’ or ‘Commit’ button in an editor to immediately close the selector window afterwards. So cancel-and-close the slot editor with a right-click on the ‘Cancel’ button, thus returning to the pilot window.

Next we are going to take a look at the schedule browser: select ‘Show schedule’ from the ‘Schedule’ menu. You can now browse the schedule in the hierarchic display, expanding and collapsing the levels by clicking on the black triangles.

When you have seen enough textual information about the schedule, close the browser and take another step.

2.12. Step 12: The Graphic Schedule Editor

Finally, let’s take a look at the graphic schedule editor! Click ‘Edit’, or select ‘Edit Schedule’ from the ‘Schedule’ menu (or the paper-and-pencil icon in the toolbar). Either of these opens another window – see Figure 5 – which allows you to see the complete schedule as a grid; slots are represented as columns, rounds as rows. Note that the columns are labeled with the host names, and the rows with the round numbers.

In each intersection of a slot column and a round row, there is a so called ‘round-slot’, which contains two frames (one for each bus channel); N-frames contain boxes in various shades of gray (yes, these are the messages transmitted in the frames), while I-frames are shown as thin gray rectangles.[24] Click on a message to select it – it will be shown in a color and listed in the ‘Selected Messages’ list at the bottom of the schedule editor window.

[23] You can also reach this summary by selecting ‘Summarize schedule’ from the ‘Schedule’ menu.
[24] See Section 11.1 for more information on frame types.

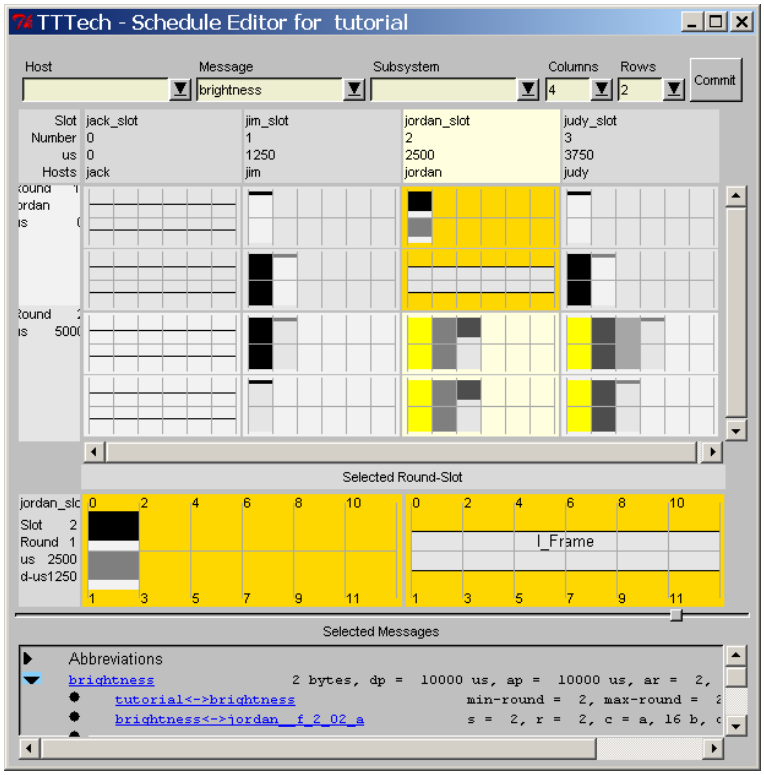


Figure 5: The schedule editor of TTP-Plan

You will note that if you select a message, all occurrences of the message are highlighted; indeed, if the subsystem that sends this message is executed by several hosts, the message would be highlighted in all these slots.[25] Section 3.10 describes in detail how to use the schedule editor.

Try displaying all occurrences of the message ‘brightness’ in this schedule by selecting this message from the drop-down list in the upper part of the window. You will see that it is transmitted in the slots of ‘jordan’ and ‘judy’, and on both channels – in other words, this message is replicated on two channels and on two hosts, and will be present in the cluster if either a channel or a host (or both) fail, introducing a high level of fault tolerance.

Close the schedule editor with the ‘Close’ button at the top right and proceed to the last slide.

[25] You can see this better if you try it with the other supplied example database, `x_by_wire`; this is a larger cluster design with replicated subsystems.

2.13. Conclusion of the Guided Tour

You have now taken a quick tour through a complete cluster design with TTP-Plan. Of course, this was a very small cluster; the size of a design (the number of hosts, subsystems, and especially messages) will be greater in a real scenario. But the steps taken stay the same.

Save your cluster design to a database file (`.cdb`) using either 'Save cluster' or 'Save cluster as ...' from the 'File' menu. 'Save cluster' saves the database in

`C:\TTP\<toolname>\<version>\tutorial.cdb`

by default, using the name of the cluster as database name. If you started with an existing database and modified it, 'Save cluster' overwrites the original. 'Save cluster as ...' allows you to choose the name and location of the file.

In the above directory you will see a few other files and directories, for example the sample databases. Although the result of this tutorial should be identical with the corresponding sample database, you may want to save it under a different file name in order not to overwrite the sample.

The subdirectory named `tutorial.ddb` contains the download files for the cluster and each of its nodes and is generated during scheduling. It is needed by TTP-Load to download the design data to the nodes. If you want to continue designing your system at the node level, start TTP-Build and load the cluster database you just saved. There you can design each node in detail.

Finally, there is the log file or *journal file*, recognizable by the extension `.cdt_journal`. It is created whenever you define a new cluster, and logs all editing operations and actions performed on the current database. See Section 4 for details on this file.

All the functions provided by the step-by-step guide are also available via TTP-Plan's menus and the 'Pilot'. The 'Pilot' page is selected by clicking on the 'Pilot' tab of TTP-Plan's main window.[26] The 'Pilot' displays TTP-Plan's object model. By double-clicking on an object or link you can activate the corresponding object or link selector, respectively.

While for a new user of TTP-Plan the guide helps to get into working with this tool, the advanced user may still find it useful, since with this guide the chances of inadvertently leaving out a design step are greatly reduced.

If, after gaining some experience with TTP-Plan, you don't want the help of the step-by-step guide anymore, you can use the 'Pilot' window directly, bypassing the guide completely.

The user interface of TTP-Plan has been carefully designed for high efficiency and little overhead; if you feel that some handling is not efficient and should be improved, do not hesitate to send your comments to https://tttech-aerospace.4me.com/.

[26] If TTP-Plan is started from the command line with an argument specifying an existing cluster database (see Section 5), the 'Pilot' page is selected automatically at startup.

3. The Graphical User Interface

This section describes the *interactive* interface of TTP-Plan, which is a graphical direct-manipulation interface. The *programming* interface is described in Section 4.

This section describes the functions available via the menu bar, the step-by-step guide, the pilot, the object and link editors, the schedule editor, and the hierarchical browser windows.

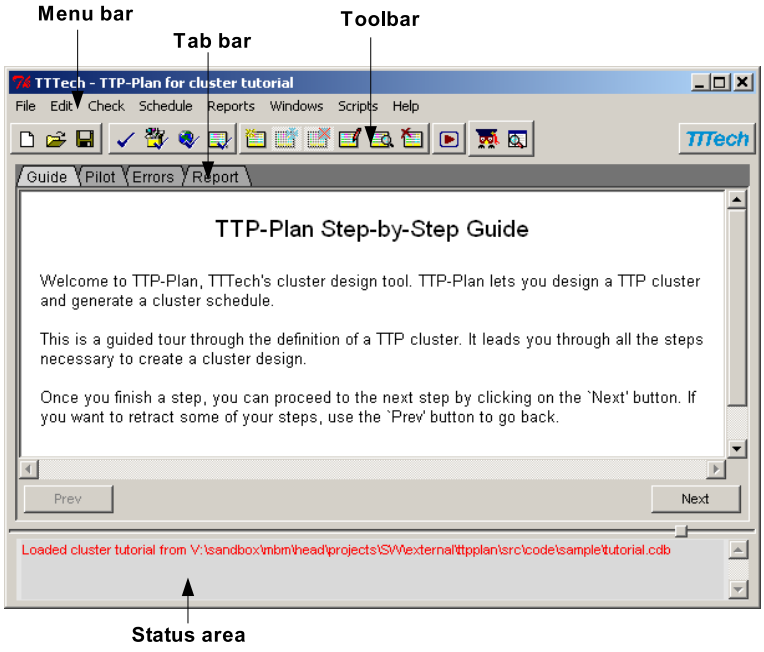


Figure 6: The main window of TTP-Plan

3.1. The Main Window

This window (see Figure 6) contains the current work area of TTP-Plan. Depending on the selection in the tab bar, either the step-by-step guide, the pilot, or a browser is displayed here. Furthermore, the main window contains the menu bar, the toolbar, the tab bar, and a status area.

The Menu Bar

The items in the menu bar provide access to the functions of TTP-Plan. Even though most of these functions can also be accessed via the step-by-step guide, access through the menu is faster and often more convenient. For a description of the different menus see Section 3.2 below or select ‘Command browser’ from the ‘Help’ menu.

The Toolbar

The icons of the toolbar represent shortcuts to the actions available from the menu bar; when moving the mouse over an icon, the command associated to the icon is displayed in a tooltip, and a description of the command in the status area. The toolbar provides the fastest way of accessing the functions of TTP-Plan.

The Tab Bar

The tab bar allows you to switch between the different pages of the main window. By clicking on a tab, the corresponding page is displayed (there is no keyboard shortcut for this action). The tab of the currently selected page is highlighted. Two kinds of pages can be distinguished here:

- *Object model* pages for editing and viewing the TTP-Plan’s object model. The most prominent of them is the ‘Pilot’, which displays the essential object model. See Section 11 for details on these pages.
- Pages for *special purposes*, such as
  - ‘Guide’ (see Sections 2 and 3.3)
  - ‘Errors’ (see Section 3.14)
  - ‘Report’ (see Section 3.2, “Reports”)

For example, the *pilot* should be visible for data input, whereas the *error browser* will be used for correcting object attributes after a schedule generation.

Note: Clicking on a tab displays the connected window, but does not execute functions in it. This is especially true for the ‘Errors’ tab: clicking on it displays the current contents of the error browser, but does *not* run any checks for errors. For this, you need to use the ‘Check’ menu or the corresponding icon in the toolbar.

The Status Area

At the bottom of the main window and the editor windows there is a gray area which is used to display help information. For example, this can be information about the currently edited attribute (in case of an editor, see Section 3.6), but also status and error messages. By dragging the horizontal line above it, the status area can be resized to display more (or less) of the information.

The contents of the status area in the main window are only cleared when the tool is exited.

3.2. The Commands

The commands of TTP-Plan can be accessed interactively via the menu bar and the toolbar, but they can also be executed from the command line and through scripts (*programming interface*, see Section 4). Descriptions of all commands are available in the 'Help' menu of the tool by selecting 'Command browser'. When selecting a command in the toolbar, the corresponding description is also displayed in the status area.

Depending on the state of the cluster design, some functions may be unavailable at specific times. In this case the corresponding item in a menu or the toolbar is dimmed. For example, 'Save cluster' in the 'File' menu will be dimmed if no cluster has been defined.

The commands of TTP-Plan are organized into groups. Each command group corresponds to a pull-down menu in the menu bar and is presented in a table in the following. The table contains the *precondition* for each command (i.e., a condition that must be fulfilled before the command can be used), an indication whether it can be executed in *batch* mode, and the possible ways of accessing it (*interfaces*). The abbreviations of the interfaces are as follows:

- **M:** Menu
- **T:** Toolbar (icon)
- **C-x:** Keyboard shortcut, where "`C`" denotes the `CTRL` key. *Note:* the second character is case-sensitive, i.e., there is a difference between `C-s` and `C-S`!

Each table is followed by a short explanation of all commands of the respective group.

File

This group provides commands for managing TTP˙Plan databases and for terminating TTP˙Plan.

Command	Precondition	Batch?	Interfaces
New cluster ...	None	Yes	MT, None
Load cluster ...	None	Yes	MT, None
Close	model-exists	Yes	M
Save cluster	model-exists	Yes	MT, None
Save cluster as ...	model-exists	Yes	MT, None
Save and exit	model-exists	Yes	M, None
Exit	None	Yes	M, None

- **New cluster ...**
  Create a cluster object for a new cluster database.
- **Load cluster ...**
  Load cluster data from a database.
- **Close**
  Close currently open cluster database.
- **Save cluster**
  Save cluster to the current database.
- **Save cluster as ...**
  Save cluster to a database with the specified name.
- **Save and exit**
  Terminate TTP˙Plan after saving any changes to the current database.
- **Exit**
  Terminate TTP˙Plan. If there are unsaved changes, TTP˙Plan will ask whether they should be discarded or saved to a database.

The ‘File’ menu also displays some recently used database files. Selecting one of these entries loads the corresponding cluster database.

Edit

This group provides commands for canceling actions, for copying text to and from the clipboard, and for starting object and link editors.

Command	Precondition	Batch?	Interfaces
Undo	can-undo	No	MT, None
Cut	cuttable	No	MT, None
Copy	copiable	No	MT, None
Paste	pastable	No	MT, None
Objects	model-exists	No	
Links	model-exists	No	
Edit delay matrix	web-and-channels-exist	No	MT

- **Undo**
  Undo last action, if possible.

- **Cut**
  Cut selection.
- **Copy**
  Copy selection.
- **Paste**
  Paste selection.
- **Edit delay matrix**
  Edit the transmission delays between the nodes, set by the Goodie `set_delay_matrix.py`.

Edit.Objects

This submenu provides direct access to the selector windows of all classes of the TTP˙Plan object model.

Edit.Links

This submenu provides direct access to the selector windows of all associations of the TTP˙Plan object model.

'Cut', 'Copy' and 'Paste' use the standard clipboard mechanisms and can therefore be used to interact with other applications. For example, you can copy object names from a spreadsheet into the input fields of an object selector when creating objects. With the 'Undo' function you undo the latest step you have taken, as far as it is reversible (e.g., remove an object you have created). You cannot "unload" a database you loaded before, or "undo" a schedule, for example. But all reversible steps can be retraced, one by one, back to the very first one. This includes even implicit functions, e.g., if you created a link that implicitly created an object, and you undo the creation of the link, the object will disappear as well. If you deleted such a link by mistake, clicking 'Undo' will restore both the link and the object.

Check

This group provides commands for checking various aspects of the TTP˙Plan object model.

The results of the checks are displayed in the error browser. The error browser provides hyperlinks to the editors of the erroneous objects or links. It can be accessed via the `Errors` tab, but is also opened automatically when errors are found.

TTP˙Plan applies these checks automatically at the right time: object-level checks during editing, global checks before making a schedule, and schedule checks after making a schedule.

Command	Precondition	Batch?	Interfaces
Everything	model-exists	Yes	MT
Object correctness	model-exists	Yes	MT
Global correctness	model-exists	Yes	MT
Schedule	is-scheduled-or-frozen	Yes	MT

- **Everything**
  Check TTP˙Plan model for object, global and schedule correctness. Note that, if an object-level check fails, the subsequent checks (i.e., global and schedule) are not performed.
- **Object correctness**
  Check TTP˙Plan model for object-internal correctness and object integrity. This command checks all defined objects and links for correctness and completeness of their attributes (according to object invariants).
- **Global correctness**
  Check TTP˙Plan model for global correctness. Global means the check covers the interactions between the different parts of the object model, as well as more general aspects of the system (system invariants), but not the object-internal settings.
- **Schedule**
  Check TTP˙Plan model for schedule correctness (only applicable when a schedule exists). This check is performed automatically after each execution of `Make new schedule`, but can also be used independently to check the effect of possibly schedule-relevant changes on an existing schedule without actually having to run the scheduler.

In the command line (batch mode), errors are displayed automatically when they are detected, i.e., during a check. Please note that, if the error is not fatal and only raises a warning, the same warning can appear several times (for example, a violated invariant that is detected both before and after scheduling will be reported each time).

Schedule

This group provides commands for generating, viewing, editing, and deleting a TTP˙Plan schedule.

Command	Precondition	Batch?	Interfaces
Make new schedule	model-exists	Yes	MT
Freeze current schedule	allow-schedule-extension	Yes	MT
Thaw frozen schedule step	schedule-is-frozen	Yes	MT
Fix message rounds	latest-schedule-is-valid	Yes	M
Edit schedule	schedule-is-valid	No	MT
Summarize schedule	schedule-is-valid	No	M
Display round-slots	schedule-is-valid	No	M
Show schedule	schedule-is-valid	No	MT
Delete schedule	is-scheduled	Yes	MT
Generate M-Hub configuration	can-generate-mhub-config	Yes	M

- **Make new schedule**
  Generate a new TTP˙Plan schedule.
- **Freeze current schedule**
  Freeze current schedule step. After freezing, `Make new schedule` and `Delete schedule` apply to a new schedule step, but not to the frozen part of the schedule. See the user manual, Section 'Incremental Scheduling', for details.
- **Thaw frozen schedule step**
  Thaw current schedule step. After thawing, `Make new schedule` and `Delete schedule` apply to the previously frozen schedule step. Thawing a schedule step automatically deletes a new schedule step scheduled after the frozen schedule step, if there is any.
- **Fix message rounds**
  Constrain all messages to the rounds as currently scheduled. This sets the attributes `min_round` and `max_round` of the `TTA.Cluster_Mode_uses_Message` links to the current rounds of the respective messages (`a_round`).
- **Edit schedule**
  Open the graphical schedule editor. There you can make changes to the schedule by drag-and-drop.
- **Summarize schedule**
  Display a summary report for the cluster schedule.
- **Display round-slots**
  Edit round-slots of schedule graphically.
- **Show schedule**
  Display the cluster schedule in a hierarchical browser.
- **Delete schedule**
  Delete TTP˙Plan schedule.
- **Generate M-Hub configuration**
  Generate an M-Hub configuration header file.

If object or link attributes are modified after making a schedule, the schedule will often become invalid.[27] In this case, all schedule commands except ‘Make new schedule’ and ‘Delete schedule’ are disabled. You can make the existing (invalid) schedule valid again by changing the attributes back to their original values, or you can make a new schedule, based on the current inputs. In any case, a valid schedule is a prerequisite for code generation.

Reports

This group provides access to various reports. The predefined reports display all instances of the selected class or association in a hierarchical view, or show other, tool-specific information. User-defined reports can be generated by means of scripts.

Reports are organized into categories. The menu displays a drop-down list for each report category; selecting an entry from the list generates the corresponding report. All reports provide commands for searching and saving in a context menu and/or as function buttons.

Command	Precondition	Batch?	Interfaces
Separate window	None	No	M
Objects	is-applicable	No	
Links	is-applicable	No	
Add ...	None	No	M

- **Separate window**
  Display each report in a separate window.
- **Add ...**
  Add a report to a category of the “Reports” menu.

Reports.Objects

Displays reports of all classes of the TTP˙Plan object model, providing detailed object information.

Reports.Links

Displays reports of all associations of the TTP˙Plan object model, providing detailed link information.

A report is a special instance of a hierarchical browser (see Section 3.12). The checkbox ‘Separate window’ allows you to choose whether a report should be displayed in the ‘Report’ page of the main window or in a new top-level window.

[27] Not all modifications make a schedule invalid; some have no influence on the schedule and will therefore only change the database, but not the schedule. Attributes which *do* make the current schedule invalid when modified are called ‘schedule critical’.

'Add . . . ' adds another report to an already existing or newly created report category. The 'Reports' menu can also be customized by means of startup scripts (see Section 4.7), and user-defined reports can be created using *report scripts* (see Section 4.6).

Windows

This menu provides quick access to all tabs of the main window, as well as to the selector, editor, and report windows that have previously been opened. The items in this menu change as new windows are opened or open ones closed; whenever a window is opened, it is entered into this menu so that it can be reopened quickly when required.

If a window is closed with the 'X' button supplied by the window manager, the window is deleted from the 'Windows' menu.

Scripts

From this menu, Python scripts (which are stored in separate files) can be executed using the 'Run. . . ' command; such scripts extend the capabilities of TTP-Plan and offer advanced users the possibility to generate powerful add-ons to the program (see Section 4).

Script categories can be added to this menu by startup scripts (see Section 4.7), but you may also write your own (so called "on-line") scripts, as described in Section 4.4.

By default, TTP-Plan provides the category 'Goodies' with several example scripts (see Appendix B for a complete overview). They are located in the 'goodies' folder of the tool's installation directory.[28]

If a script has been run using the 'Run. . . ' command, its name is added as a menu item, so that it can easily be run again. The output of a script, if any, is written to the status area of the main window of TTP-Plan.[29] To read more lines of output, you can use the scrollbar at the right side of the status area, and/or resize the status area by moving the horizontal line above it.

Help

This group provides information about various aspects of TTP˙Plan, as well as access to the user manual and the release notes. Also a bug report containing all necessary information can be filed from here.

[28] It might be a good idea to save your own scripts to the same directory, to allow for quick access later on.
[29] Unless the script uses specific output methods, like opening or writing to a file.

Command	Precondition	Batch?	Interfaces
About	None	No	M
Attribute browser	None	No	MT
Command browser	None	No	M
Guide	None	No	MT
Bug info ...	None	Yes	M
Release notes	None	No	M
Manual	None	No	M, None

- **About**
  Display information about product version and interface compatibility.
- **Attribute browser**
  List all attributes of the TTP˙Plan objects and links in a browser. The browser window also provides the possibility to search for a specific attribute or save selected information to a file.
- **Command browser**
  Display information about all commands of TTP˙Plan in a browser. The browser window also provides searching and saving functions.
- **Guide**
  Display Step-by-Step Guide.
- **Bug info ...**
  Write information necessary for a bug report into a file.
  This command saves all relevant information about the product, the used versions, plugins and plugin versions, as well as system and internal application (status) information, in a text file.
- **Release notes**
  Show release notes of TTP˙Plan.
- **Manual**
  Open the TTP˙Plan user manual with the default PDF viewer.

If any plugins are installed for TTP-Plan, they are displayed in the ‘About’ window.

Note: the *attribute browser* includes even attributes of objects that are outside the main scope of the tool, but are used by it internally. These objects cannot be edited and are not accessible via ‘Edit.Objects’.

3.3. The Step-by-Step Guide

This feature, which can be very helpful in finding a regular working scheme with TTP-Plan, is described in the tutorial of this manual (see Section 2). The guide takes the user on a guided tour through the steps of a cluster definition and offers one possible (recommended) sequence of creating and editing the objects and links of a cluster. It can be accessed via the ‘Guide’ tab or by selecting ‘Help.Guide’ from the menu.

The guide is opened automatically if TTP-Plan is started without parameters, i.e., without initially loading a database. Each step is explained in the guide window and can immediately be carried out by the user. This is similar to a typical 'wizard' applet of other design tools. However, the guide does not control, or keep track of, the user's actions. The user may freely divert from the actions suggested by the guide without any objections by the program, like selecting the pilot and creating new objects at times where the guide does not suggest this. In this respect, the guide is more user-friendly (and less arrogant) than the typical 'wizard'.

Using the guide, especially going through the tutorial with the help of the guide, is recommended if you are new to TTP-Plan. After one or two trials, the object model and tool handling will become more familiar.

3.4. The Pilot

The pilot displays the essential object model of TTP-Plan (see Figure 17 in Section 11). It can be accessed by clicking on the 'Pilot' tab in the tab bar. Using the pilot, the objects and links of TTP-Plan can be selected for editing, creation, or deletion. Left-clicking on any of the graphic items in the pilot window displays any subclasses this class or association may have; selecting one of them opens the appropriate *selector* (see Section 3.5). If there are no subclasses to a class, the selector is opened by double-clicking on the item.[30]

Once there, new objects or links can be created from scratch or as copies of existing objects or links, and they can be edited or deleted. The pilot itself cannot be closed, but you can switch to any other view by selecting it in the tab bar.

3.5. The Editors

All object classes and associations of TTP-Plan share a common editor layout, where only the actual object attributes differ. Each editor has two windows, of which only one is active at any given time:

- The *selector* window (see Section 3.6) lists all existing instances of a class or an association – e.g., all hosts, or all links of `TTA.Subsystem_sends_Message` – and allows the viewing of their attributes. It also enables the user to create, copy, modify, and delete these objects or links, unless they are read-only.
- The *attribute editor* window (see Section 3.9) shows the attributes of a single object or link. If that object or link is not read-only, the attribute editor allows the values of the required and optional attributes to be changed.

By double-clicking on an object in the pilot, the object selector for the corresponding class is opened, whereas clicking on a hyperlinked object name – for example, in a browser – displays the attribute editor for this particular object directly.

[30] If a class has – and allows – only exactly one (active) instance (like the '`Cluster`' object), the attribute editor is immediately opened as well.

3.6. The Object Selector

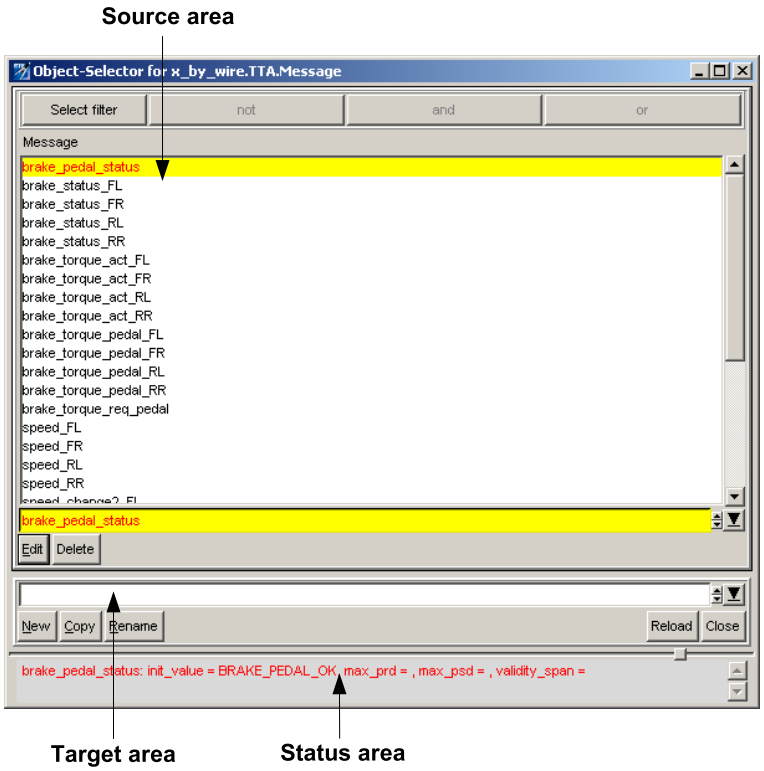


Figure 7: The object selector

Depending on whether the selector lists objects or links, it has one or several columns for names. In an *object* selector (see Figure 7), the names of all objects of the selected class are listed, e.g., in the `Host` selector, all `Host` objects are listed.

In a *link* selector, there are two (or three) columns of object names, depending on the number of roles in the association (see Section 3.8).

The selector window is subdivided into three areas. Looking from top to bottom, you see:

- the source area,
- the target area, and
- the status area.

The Source Area

The source area is used to select existing objects for editing, deleting, copying, or renaming. It comprises a filter button box, a list box showing all existing objects (i.e., all objects

satisfying the selected filter), an input field showing the selected object, and a button box for object manipulation.

The filter button box of the source area can be used to restrict the number of objects shown in the list box. Clicking on the 'Select filter' button pops up a menu with the filters applicable to the object type in question. Selecting one entry of this menu restricts the set of objects to those satisfying the corresponding filter condition. The text of the 'Select filter' button changes to indicate the chosen filter. The 'and', 'or', and 'not' provide the possibility of modifying the selected filter condition.[31] Moving the mouse over the 'Select filter' button displays a description of the selected filter condition in the status area (see below).

The list box and the input field are synchronized, i.e., selecting an object in one automatically selects the same object in the other. The input field provides completion; entering an incomplete object name and pressing `<Alt-i>` completes the incomplete name as far as possible; if no further completion is possible, `<Alt-i>` uses a pop-up menu to show the alternatives matching the incomplete name. You can then either select a completion from the menu or type more characters and use completion again.

Some more hints for keyboard aficionados: the tabulator key moves the focus between the input fields, the fastest way to clear an input field is the key combination `<Home>` `<Ctrl-k>`, the fastest way to enter a new value into a field when the cursor is already at the end of the field (e.g., from previous typing) is `<Shift-Home>` (this highlights the whole field contents) and typing the new value.

The object manipulation buttons always act on the selected object, i.e., clicking the 'Edit' button opens the attribute editor for the selected object, clicking the 'Delete' button deletes the selected object.[32] *Read-only* objects and links are usually generated automatically by TTP-Plan during each scheduling run, hence it would not make sense to edit them manually. In this case the 'Edit' button is labeled 'View' and there is no 'Delete' button.

The Target Area

The target area is used to select the name of a new object for object creation, copying or renaming. It comprises an input field showing the new name and a button box.

The input field allows navigation by the names of existing objects. This is helpful if many objects have similar names – you can select an existing name and edit it to derive a new name. This entry also allows completion. The functions of the buttons can be described as follows:

- 'New' – Creating a new object from scratch.
  Enter the name of the new object in the target input field and press the 'New' button. An empty object (with default values for optional attributes) is created and added to the object list (except you cannot create a new cluster here).

[31] These buttons are disabled unless a filter condition is selected.
[32] 'Delete' doesn't ask if you are serious – it just does what you tell it to do!

- ‘Copy’ – Creating a new object by copying an existing one.
  Select the existing object by clicking on its name in the object list, or enter the name manually into the source input field. Then enter a name for the new object into the target input field and press the ‘Copy’ button. A new object with the same attribute values as the selected one is created and included in the object list (not possible for the current `Cluster`).
- ‘Delete’ – Deleting an existing object.
  Select the object or enter the name in the source input field; then click the ‘Delete’ button (not possible for the current `Cluster`). *Note:* although the selected object and all its links to other objects are deleted, they can easily be recovered by using the ‘Undo’ function (see Section 3.2, ‘Edit’).
- ‘Rename’ – Renaming an existing object.
  Select the object to be renamed, enter the new name in the target input field, and click ‘Rename’. If you entered a new name that is already present in the database, a ‘Name Clash’ will occur. *Note:* you cannot create a new object with this operation, only rename an existing one. Also you cannot rename the current `Cluster` object.
  Although you can rename a ‘Host’ object, this will make an existing node database of that host invalid.
- ‘Reload’ – Updating the object list of the source area.
  If you created an object of this class somewhere else (e.g., implicitly in a link editor), clicking the ‘Reload’ button resynchronizes the object list.
- ‘Close’ – Closing the selector window.
  Clicking the ‘Close’ button hides the selector window. It is not removed from memory, though, and can quickly be accessed again via the ‘Windows’ menu or by double-clicking on the object in the pilot.
  However, if you close an editor window by clicking on the *window manager’s* ‘Close’ button, the window will be deleted from the ‘Windows’ menu. In this case, it will take slightly longer to display that editor again.

The last button activated becomes the“default” button – just pressing ‘Enter’ will activate the same button again. The default button is shadowed. So, if you want to create a number of objects, enter one name, click ‘New’, enter another name and just press ‘Enter’, next name, ‘Enter’, . . . You don’t need to grab the mouse each time.

The Status Area

The status area is used to display information about the buttons and about the selected object. This area can be resized with the mouse by dragging the horizontal line above it.

Moving the mouse over any of the editor buttons displays a help message about the button’s function in a tooltip (for currently inactive buttons in the status area). If you click on a button without having first entered the name(s) of the object(s) to process, the status area will display a message asking you to provide the necessary information.

Some object types provide information about the selected object. For such a type, selecting an object displays some attributes of interest in the status area. For instance, the

`TTA.Msg_Type` editor will display the ‘length’ and some other attributes of the selected message type.

3.7. Object Naming Conventions in TTP-Plan

Naming conventions for objects in TTP-Plan apply to all objects, whether the object names are entered manually or created automatically (as they are for slots and frames).

Objects may have any name that constitutes a valid identifier in C. Object names can have (virtually) any length, are case-sensitive, and may only contain upper- and lowercase characters, digits, and the underscore sign (useful as a separator, like in `sounds_nice`).

Other characters are not allowed and will immediately be rejected with an “Invalid Name” exception.

It is advisable to use self-explanatory names with not more than 10 characters, especially on screens with lower resolution (less than 1280x1024).

No two objects in a cluster database can have the same name (not even if they belong to different classes!). Such an input will be rewarded with a “Name Clash” exception.

3.8. The Link Selector

Links are a sort of objects, too, but they require special handling; their main attributes are the objects they link, but there may be ‘real’ attributes in a link as well (see Section 3.9). For example, a link of the association `TTA.Host_runs_Subsystem_in_Cluster` links a host to a subsystem and the cluster, and thus defines the distribution of subsystems in the cluster.

The *link selector* for a specific association is opened by double-clicking on the association in the pilot. It is quite similar to the *object selector* (see Section 3.6), but exhibits two specialties (see Figure 8):

- Instead of a single list box, there are as many as there are roles in the association. Accordingly, there are several input fields in both the target and the source area as well.
- In addition to the filter button box for the source area, there is also a filter button box for each of the target input fields.

Double-clicking on a link, or selecting it and clicking ‘Edit’ opens the *attribute editor* for that link (see Section 3.9).

In addition to the operations already described for the object selector, there are some actions that only make sense for links. These are described in the following.

Restricting the Link List

Since there may be a large number of links of a certain class in a database, you may want to see only a subset of all these links.

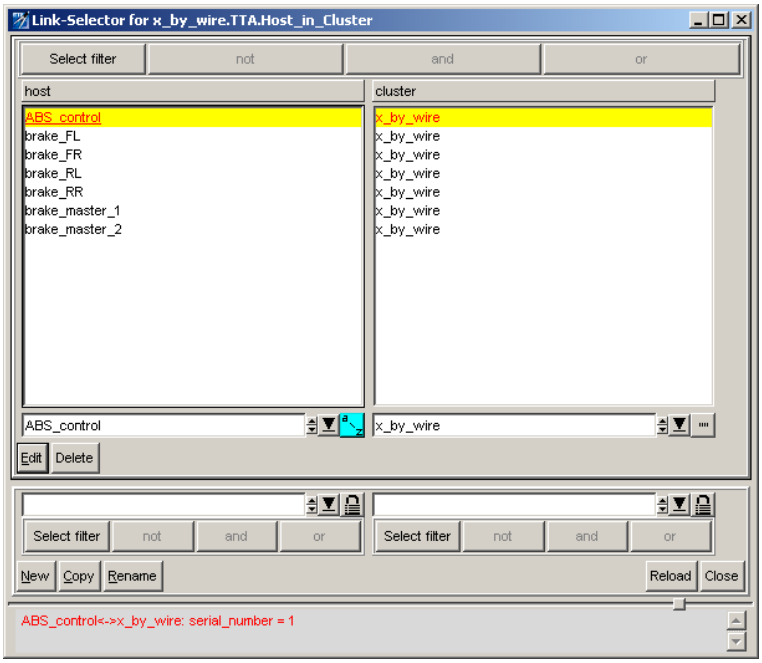


Figure 8: The link selector

For example, when working on a subsystem `Su2`, you may only want to edit the links `TTA.Subsystem_sends_Message`

that concern `Su2`. So you *restrict* the complete list of links to those which link `Su2`, by setting a lock on the subsystem's source input field:

Select any link that links `Su2`, or enter `Su2` in the source input field in the 'Subsystem' column, click on the graphic button (which currently either shows 'a-z' or '···'), select 'toggle lock', and the link list is restricted to those that link `Su2`.

Locking the Target Input Fields

There is a 'lock' function for the target input fields as well. Its function is different, though: if a lock is set on a target field (which is the default state for all target fields), the field will not automatically take the value from the corresponding source field. If a target field is unlocked, selecting an object in the source field will automatically copy the object's name into the corresponding target field. This feature is useful for copying several links in a row.

Sorting the Link List

A link list can be sorted by the name of any of the objects it links; for most links, this means a choice of two sort keys, for ternary associations like `TTA.Host_runs_Subsystem_in_Cluster` there are three possible sort keys.

Initially, the first column of the link list is selected as the sort key, as can be seen from the blue graphic action button showing an 'a-z' picture. Other object columns are selected as sort key by clicking on the action button of that column and selecting 'sort by' from the context menu. The original sort key column will then lose the 'a-z' picture, since there can only be one sort key at a time.

The column to be used as sort key can also be selected by clicking into the column label.

3.9. The Attribute Editor

Select an object in the selector and click 'Edit'.[33] This switches from the selector to the attribute editor – often simply called "Object editor" – where you can look at all the attributes and change the required and optional ones, unless the object is read-only (in this case, the 'Edit' button of the selector is labeled 'View').

The attribute editor is subdivided into three areas (see Figure 9):

- the *entry area* to the left
- the *error area* to the right
- the *status area* at the bottom

You can change the relative widths of these areas by moving the separating line with the mouse.

The *entry area* shows all the attributes of the object. Each attribute is characterized by a name and a value. Clicking into the input field of an attribute displays an explanation for the meaning of the attribute in the *status area*. Pressing the *F1* key displays an *attribute browser* with help information on this and the other attributes of the object you are currently editing.

Many objects and links have an attribute titled `description`, which can be used for documentation purposes. The button with the three dots next to the entry field allows "further editing" of the field contents – if you click on it, a larger editor window pops up, allowing you to enter a longer description.

Some attributes allow for the selection of values that are in fact objects themselves and can be viewed and/or edited. This is indicated by a table-like button beside the input field of the respective attribute. Clicking on this button opens the editor of the associated object.

Note: the editable attributes may have default values (displayed directly in the corresponding input field) or the possibility to have their values computed during scheduling. For the

[33] If you double-click on the class '`Cluster`' in the pilot, you will immediately enter the attribute editor, i.e., 'Edit' mode. This is a useful shortcut operation because each database can only contain one active `Cluster` object. Therefore, you cannot create a new one now.

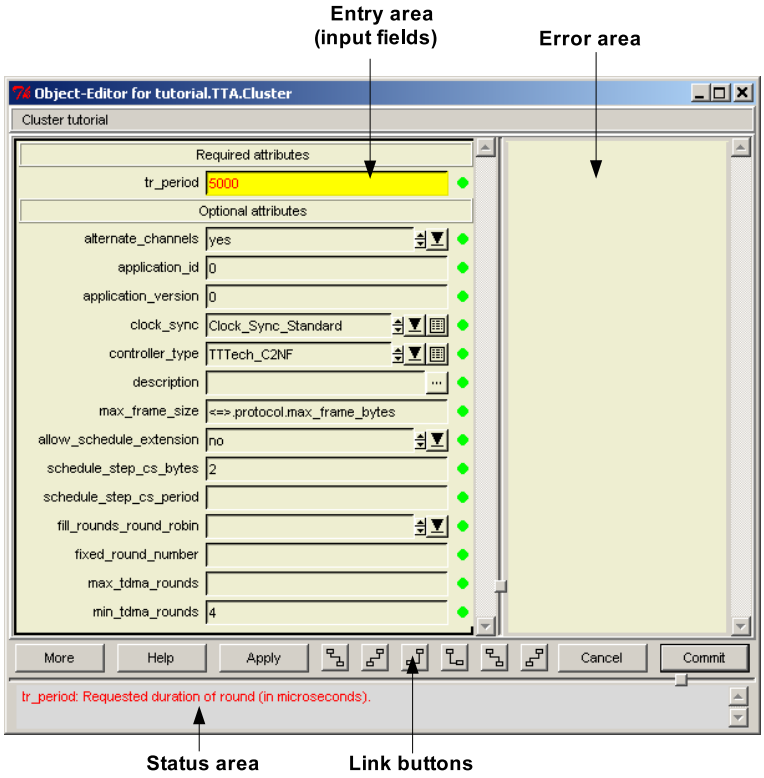


Figure 9: The object editor

latter ones, the currently assigned value is displayed in parenthesis in the status area – along with the attribute description – as long as the input field is empty, i.e., no value has been entered, imported or calculated. Please note that this value is preliminary and may change during an editing session because it is computed at scheduling time! It is mainly intended for debugging and may also differ between different versions of TTP-Plan. If the calculation of an attribute is not possible (for example, because there is more than one choice), the parenthesis in the status area is empty. In this case you need to specify the value of the attribute manually. Typical example: when there are several installed plugins for the same layer, TTP-Plan cannot determine the active scheduler and code generator.

The input fields of all changeable attributes provide a history feature that allows you to quickly access the values previously typed in each field: use `<Alt-Up>` and `<Alt-Down>` go through the different values entered during the current execution of TTP-Plan. `<Alt-q>` shows a menu with all values entered for this field. Input fields with drop-down list boxes also support completion (see Section “Source Area”).

If you enter an erroneous value into an input field, the *error area* will display one or more error messages. Clicking on an error message puts the focus into the erroneous attribute.

When there are error messages, putting an erroneous input field into focus will highlight all error messages complaining about that field's value.

The attribute browser features several buttons between the entry/error area and the status area:

- When the editing of the various attribute input fields is finished, click 'Commit' to store the inputs, or 'Cancel' to undo all changes to this object since the last commit operation. Either one takes you back to the selector window, where another object can be selected for editing. Note that committing is not possible as long as there are error messages – you must correct all errors before you can commit.
- Clicking on the 'Apply' button stores (commits) the inputs without leaving the attribute editor.
- 'More' displays the internal (view-only) attributes below the required and optional ones.
- 'Help' opens an attribute browser containing all attributes of the current object, grouped by types (required, optional, etc). The 'F1' key opens the same browser, but jumps directly to the currently selected attribute.
- Clicking on one of the graphic link buttons next to the 'Apply' button takes you to a link editor (see Section 3.8). The link editor associated to a certain button is displayed when you move the mouse cursor over the button.

The attribute editor of a *link* is almost identical to that of an object, except that there are no link buttons (one link cannot be part of another link).

Entering Symbolic Expressions

In the attribute editor you can also enter symbolic expressions as attribute values. For instance, the entry for an integer attribute will accept a literal number as well as a symbolic expression corresponding to a number. In many cases, the use of symbolic expressions is highly advantageous. Symbolic expressions can refer to attributes of other objects or links in several ways, specifically:

- The cluster attributes are available as

  ```
  TTA.Cluster.attribute-name
  ```

- The attributes of other objects are available as

  ```
  <object-name>.attribute-name
  ```

  where `<object-name>` must be the name of an already existing object and `attribute-name` the name of an attribute of that object (see Section 12 for a description of the attributes of the different classes and associations).
- The attributes of links are available as

  ```
  <assoc-name,left-name,right-name>.attribute-name
  <assoc-name,left-name,middle-name,right-name>.attribute-name
  ```


where <assoc-name> specifies the association, and <left-name>, <middle-name>, and <right-name> specify the names of the associated objects that define the link.

If you want to refer to an attribute of another link of the same association, you can use '=' in place of <assoc-name>, and also in place of role names. For instance, if one defines a link between cluster cl1 and message m1, one can refer to m2's d_period in cl1 with the expression

```
<=,=,m2>.d_period
```

The following contrived example for symbolic cross-references shows some of the possibilities:

```
TTA.Cluster.tr_period - <msg_x>.validity_span
                  + <Cluster_uses_Message,cl,msg_y>.period
```

3.10. The Schedule Editor

After all hosts, subsystems, messages etc. have been defined, all links between them created, and a schedule has successfully been generated, you will usually want to have a look at the schedule as well. There are two possibilities: a *textual* representation, accessible by selecting 'Show schedule' from the 'Schedule' menu (see Section 3.13 for details), and the graphical *schedule editor*, which gives a better overview and is described in detail in this section. You can open it by selecting 'Edit schedule' from the 'Schedule' menu.

The Display of the Schedule Editor

The schedule editor (see Figure 10) consists of four parts (from top to bottom):

- the object selection drop-down list boxes,
- the schedule grid,
- the magnifier window for the current round-slot, and
- the list of selected messages/message boxes.

In the schedule grid, the complete schedule is graphically accessible. (Generating this grid may take some seconds for huge clusters). All slots of the cluster cycle are displayed as columns, in the label rectangle at the head of the column you can find the name and number of the slot, its start time (only for the first round), and the name of the host sending in this slot.

The rows stand for the individual rounds of the cluster cycle; in the label rectangle at the left, the round number and its start time (i.e., of its first slot) within the cluster cycle are displayed. By clicking on any label rectangle, the slot column or round row is highlighted for better readability of large schedules. Selecting a host in the 'Host' drop-down list box highlights all slots where this host sends.

Within each intersection of a slot column and a round row – we call this a round-slot – there are two frames. These are the frames which will be transmitted on the two channels in this slot and round. The currently selected round-slot is highlighted and also shown in the magnifier window below the schedule grid. The sending host is displayed in the round header.

Clicking into some free space within a round-slot selects the whole round-slot; clicking directly on a frame/message within a round-slot selects this frame/message (and also adds it to the 'Selected Messages' list, see below), but the round-slot itself will not be selected.

Each frame is displayed as various boxes of gray – these represent the individual messages which are packed into the frames. The messages (message boxes) in the frames are also displayed in the magnifier window, since they may be quite small in the schedule grid (sometimes even too small to be clicked on). "Overfull" frames are displayed as such and are marked red for easy recognition.

Messages in the Schedule Editor

Clicking on a message, or selecting its name in the 'Message' selection drop-down list box at the top of the window will *select* this message; all occurrences of it will be colored in the schedule grid and the magnifier window, and the message will be added to the 'Selected Messages' list at the bottom of the window. This list can become long, so you can resize the list area by dragging the horizontal line above the list area with the mouse. *Note:* if messages are placed in message boxes, you can only select and view the boxes as such, but not the individual messages.

The item 'Abbreviations' in the 'Selected Messages' list can be expanded by clicking on the black triangle, to show explanations of the abbreviations used for the displayed message parameters.

Selecting a message is useful to see in which slots/rounds it has been scheduled for transmission. Several messages (currently up to 16) can be selected at one time to see interdependencies easily – they are automatically highlighted in different colors. Additionally, if you select a message in the drop-down list box and the schedule grid is wider than the window, the window will scroll to the first slot where the message is transmitted.

To deselect a selected message, you can click on it again, or right-click on its name in the 'Selected Messages' list. To deselect all currently selected messages, point your mouse cursor on the first message in the 'Selected Messages' list and click the right button repeatedly until the list is empty.

If you select a subsystem by employing the 'subsystem' selection drop-down list box at the top of the window, all messages generated by this subsystem are highlighted (up to the maximum number).

Selecting a host in the 'Host' selection drop-down list box will not select any messages, but highlight the appropriate slot columns in the schedule grid and scroll to the first one if the schedule grid is wider than the window. In this way, the sending slots of a host can be found quickly if there is a large number of hosts present.

The Schedule Editor for Multiweb Clusters

For clusters with several webs – so called *multiweb clusters* – the schedule editor displays all channels of all webs (see Figure 11). This means that in each round-slot you can see one frame per web and channel.

Dragging Messages into Different Frames

TTP-Plan allows graphical schedule editing by drag and drop: you can drag a message from its current position (frame or round) to another. With this action, the attributes `min_round` and `max_round` of the corresponding `TTA.Cluster_uses_Message` link are set to the current round of the message (`a_round`). In this way, you can optimize the current schedule and generate shorter slots, thus allowing for shorter overall rounds. You can even try to find a feasible schedule if the automatic scheduler of TTP-Plan failed to do so. For optimizations, it is also possible to drop messages into frames that are already (over)full; messages that overfill a frame will be displayed in shades of red, as opposed to the usual shades of gray, to show that this is not a correct frame. Of course, such overfull frames must be resolved again to make a valid schedule.

However, depending on the attributes of the message, you might not be able to:

- drop messages into rounds where their period or phase constraints would be violated (yes, this is calculated online, while you drag the message)
- drop messages on I-frames
- move replicated messages to a round-slot where there is an I-frame on one of the channels (the message needs to be non-replicated for this, since the I-frame cannot transmit messages)
- move messages out of their slot (if you want another host to transmit this message, you must link the subsystem that sends this message to the other host instead of the current one, and make a new schedule)
- move messages within the frame (there should never be a need for this).

When you are satisfied with the result, you should close the schedule editor and select ‘Fix message rounds’ from the ‘Schedule’ menu. If you do this, it forces the current messages into their rounds and will keep them there, even when you add new messages. TTP-Plan will remember the rounds into which the messages were dropped, and future scheduling runs will put them back there. If you don’t run ‘Fix message rounds’, making a new schedule for this cluster will destroy all message editing operations in the schedule editor!

Dragging Slots to Change the Round Layout

You can also change the round layout by moving slots within the round. Just click on the label rectangle at the top of a slot column, hold the mouse button down and drag to the left or right; when the mouse gets close to the edge of another slot, a bright yellow line

will light up, indicating that you can ‘drop the slot here’. If you do so, the slot will move into its new position within the round and the schedule grid will be redrawn.

This change is immediately recorded in the cluster database (it is reflected in a host attribute called `slot_sort_key`) and you do not need to run ‘Fix message rounds’ or any other command to commit this change.

Editing Large Schedules

When working with large clusters, the ‘columns’ and ‘rows’ drop-down selection list boxes will prove useful; here the display area of the schedule grid can be set. If you have 40 hosts, but only want to see 10 slots at a time, select ‘10’ columns. The schedule grid will be redrawn, and the frames will be much larger, better to see and to click on.

Selecting a value can be performed in two ways:

- Open the drop-down list and click on the desired value; the schedule is immediately redrawn accordingly.
- Click into the input field, enter the desired value with the keyboard, and press ‘Enter’.

The same works for rows if your schedule has many rounds; in this case, selecting a smaller number of rows will give a larger display. If there are more slots and/or rounds than are currently displayed, you can always use the vertical and horizontal scrollbars of the schedule grid.

When selecting a host or a message from the selection drop-down list box, the display will immediately scroll to the first sending slot of this host/message.

3.11. The Round-Slot Viewer

After the successful generation of a schedule, you may want to open the round-slot viewer and have a look at the schedule timing. Select ‘Display round-slots’ from the ‘Schedule’ menu to do this.

The round-slot viewer consists of four parts (from top to bottom):

- the object selection drop-down list boxes,
- the schedule grid,
- the magnifier window for the current round-slot, and
- the description window.

In the schedule grid, the complete schedule is displayed graphically (see Figure 12). Generating this grid may take some seconds for large clusters.

The slots of the round are displayed as columns. The label at the head of the column shows the name of the host sending in this slot, the slot number, and the start time of the slot (in the first round).

The rows display the rounds of the cluster cycle. For multiweb clusters, all channels of all webs in the cluster are displayed, even if the current host does not participate in all of

them.[34] The label at the left shows the round number and the start time of the round (i.e., of its first slot) relative to the cluster cycle.

By clicking on any label, the associated column (slot) or row (round) is highlighted.

Each intersection of a slot column and a round row displays a round-slot containing several objects.

At the top of the magnifier window, the slot time is displayed for both channels (first channel above, second one below). The time is split into four parts that are equal for both channels (from left to right):

- *Transmission phase*: the time span needed for transmission of the frames. I-frames and N-frames are displayed in different colors. Overfull N-frames are displayed in red color to highlight them.
- *Post-receive-phase (prp)*: the time span immediately after transmission phase, during which certain services are performed.
- *Idle time*: this time is needed to stretch the durations of the slots to meet the specified round duration. This idle time is unused bandwidth.
- *Pre-send-phase (psp)*: the time span immediately before action time, during which frame transmission is prepared.

  The sum of prp, idle time and psp determines the inter-frame gap (IFG). It is limited by the slowest controller in the cluster.

Below the slot time, the user interrupts are displayed; again, for all channels of all webs in the cluster. User interrupt 1 is always shown on top, user interrupt 2 below. The two interrupts are displayed in different colors, and the ID and web of each interrupt are indicated.

Clicking into a round-slot selects it. The currently selected round-slot is highlighted and also shown in the magnifier window below the schedule grid (titled ‘Selected Round-Slot’). The magnifier displays this round-slot larger than in the schedule grid and shows some additional information:

- The kind of each item in the round-slot.
- A time grid showing the time from the beginning of the cluster cycle.

The bottom part of the window holds the description window, named ‘Current Round-Slot’. This window contains the following information about the selected round-slot (click on the black triangle to the left of an item to expand it and view the details):

- Usage of this editor.
- The abbreviations of the displayed attributes
- Information about the selected round-slot.
- Information about the frames contained in the selected round-slot.
- Information about the messages of the N-frames, if there are any.

You can resize the description window by dragging the horizontal line above the description window.

[34] From top to bottom: channel 0 of web 0, channel 1 of web 0, channel 0 of web 1, channel 1 of web 1, etc.

Selecting Frames

Selecting a name in the ‘Message’ entry field at the top of the window will highlight all frames containing this message in a single color.

Selecting a name in the ‘Subsystem’ entry field will highlight all frames containing the messages of this subsystem in one single color.

Pressing the return key in a message or subsystem entry field will toggle the selection of the corresponding frames. Selecting another message or subsystem automatically deselects the previously selected message/subsystem, if any.

Selecting a host in the ‘Host’ entry field will not select any messages, but change the displayed user interrupts to those of the selected host. A host can only be deselected by selecting another host.

Large Schedules

When working with large clusters, the ‘columns’ and ‘rows’ entry fields will prove useful; here the display area of the schedule grid can be set. If you have 40 hosts, but only want to see 10 slots at a time, select ‘10’ columns. The schedule grid will be redrawn, and the frames will be much larger, better to see and easier to select with the mouse.

Selecting a value can be performed in two ways:

- Open the drop-down list and click on the desired value; the schedule is immediately redrawn accordingly.
- Click into the entry box, enter the desired value with the keyboard, and press ‘Enter’.

The same works for rows if your schedule has many rounds. In this case, selecting a smaller number of rows will give a larger display. If there are more slots and/or rounds than are currently displayed, you can use the vertical and horizontal scrollbars of the schedule grid to show a different part of the schedule.

When selecting a host from the ‘host’ entry field, the display will immediately scroll to the first sending slot of this host.

3.12. The Hierarchical Browsers

TTP-Plan makes use of hierarchical browser windows for several purposes, e.g., for displaying error messages, schedule information and reports, as well as attributes and commands (see Section 3.2). A hierarchical browser contains text organized as a hierarchy of nodes (similar to a directory browser, but not restricted to files and directories). A non-terminal node can be open (expanded) or closed (collapsed). Collapsing a node hides all internal information associated to the node and all its sub-nodes.

A terminal node is marked with a black bullet, a non-terminal node is marked with a black triangle. A triangle pointing to the right indicates a collapsed node, a triangle pointing downwards indicates an expanded node. Clicking on the triangle with the left mouse

button switches between the open and closed state of the corresponding node. Using the left mouse button to expand a node influences the node's state just one level deep – the sub-nodes of the node are not expanded. If you expand a node by using the right mouse button, the node's sub-nodes will also be expanded (i.e., two levels are expanded).

You can also expand and collapse nodes using the keyboard, by moving the input focus into a node and pressing `<Insert>` to expand or `<Delete>` to collapse the level. `<Ctrl-Insert>` completely expands the currently selected level and all its sublevels recursively – this may take some time for a large browser, though.

A browser can also contain hyperlinks to objects/links. Clicking on such a hyperlink jumps to the appropriate editor and starts editing the object/link in question.

All browsers provide several additional functions in a context menu available upon right-clicking. If the browser is opened in a separate window, they are also accessible via buttons at the bottom of the window (see Figure 13):

1. **'File'**
   - 'Save': opens a dialog to save the contents of the browser into an ASCII/text file. This file can then be printed using any text editor or formatting tool available on your system (for example, `WordPad` or `a2ps`). The nodes are saved in the same state as displayed. To save all nodes in the fully expanded state, use 'Edit.Expand All' first. Remember that you can expand a level completely (i.e., expand the level and all sublevels recursively) by setting the input focus on it and pressing `<Ctrl-Insert>`.
   - 'Close' (in separate window only): closes the browser window.
2. **'Edit'**
   - 'Expand All': expands all nodes of the report (at all levels). *Note:* for large structures, this may take a long time!
   - 'Find': lets you search for text matching a specific pattern. If a match is found in the contents of a collapsed node, that node will automatically be expanded to show it. The current match found will be emphasized with a light blue background.
   - 'Find next': searches for the next occurrence of the latest search pattern.
   - 'Find previous': searches for the previous occurrence of the latest search pattern.

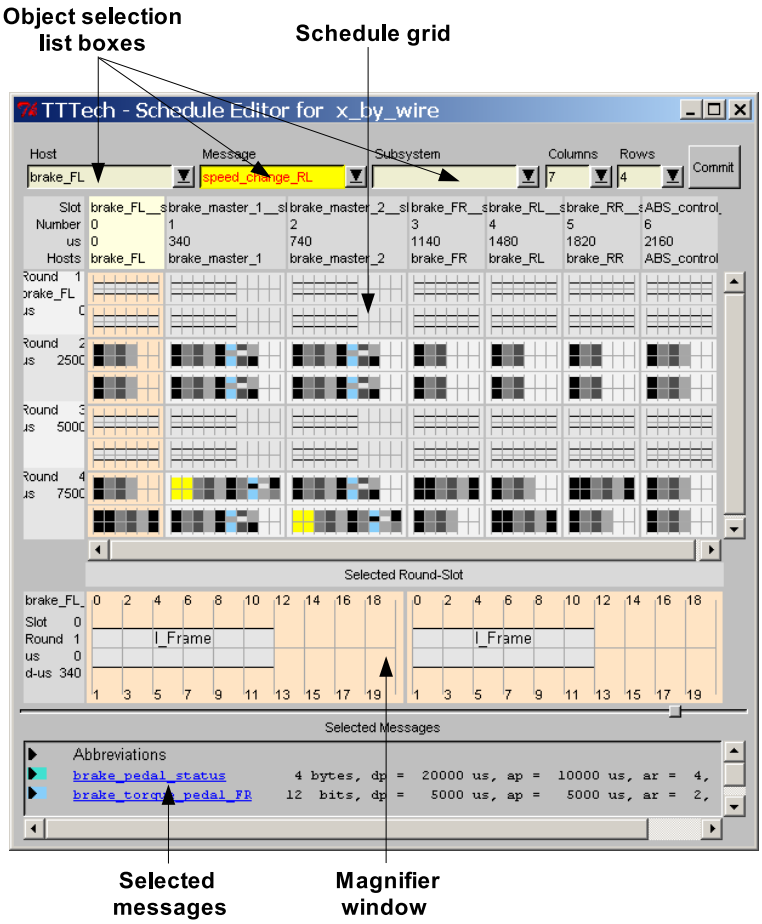


Figure 10: The schedule editor

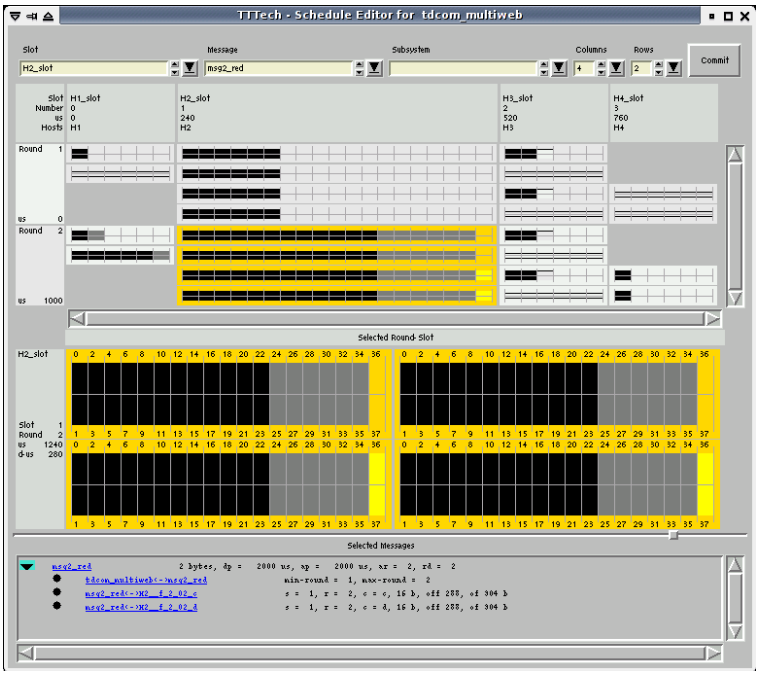


Figure 11: The schedule editor for a multiweb cluster

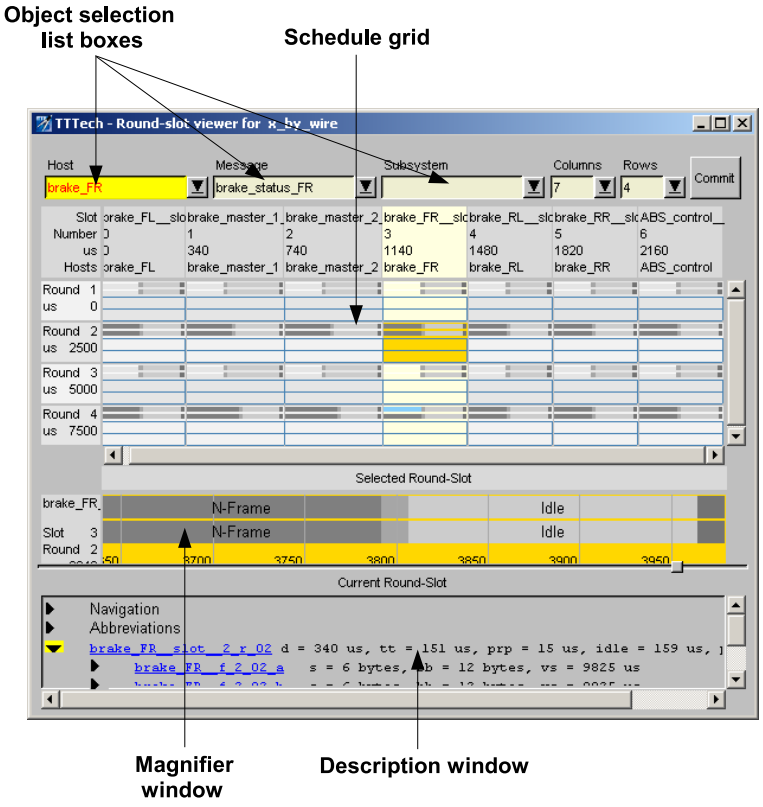


Figure 12: The round-slot viewer of TTP-Plan

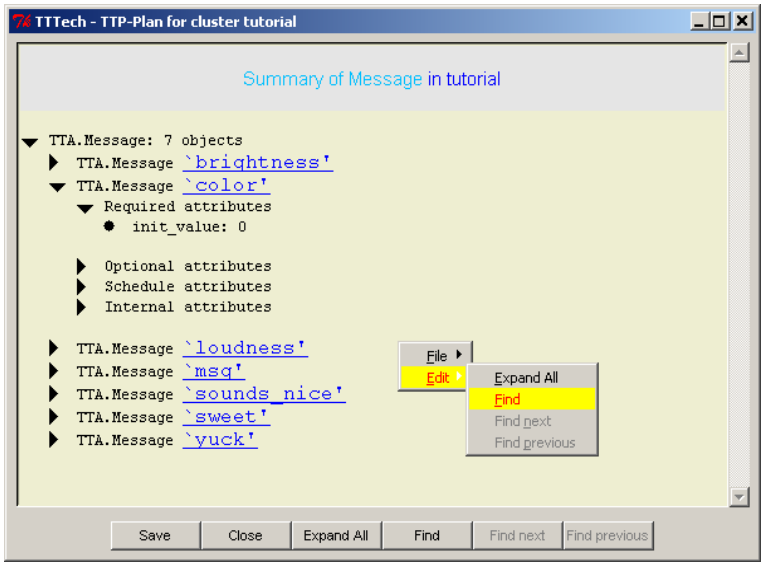


Figure 13: Browser window with context menu

3.13. The Schedule Browser

The schedule browser is one specific instance of a hierarchical browser (see Section 3.12). When you select ‘Show schedule’ from the ‘Schedule’ menu, a new window pops up and displays information about the cluster schedule.

While the graphical schedule editor (see Section 3.10) allows comfortable editing, sometimes the textual representation of a schedule offers faster access to the attributes you are interested in. So after making a schedule, select ‘Show schedule’ instead of ‘Edit schedule’ from the ‘Schedule’ menu and have a look at the schedule browser.

The schedule is displayed in a hierarchical view containing all the objects in the schedule as clickable hyperlinks (see Figure 14).

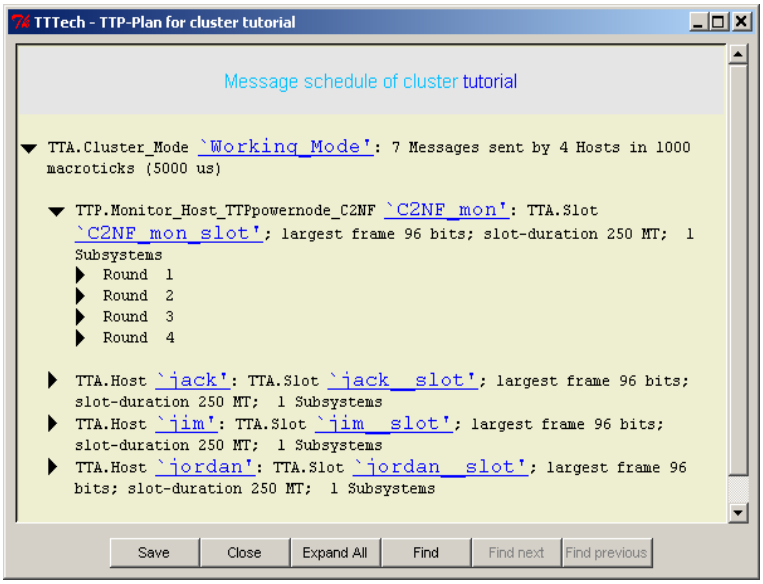


Figure 14: The schedule browser of TTP-Plan

3.14. The Error Browser

The error browser is also an instance of a hierarchical browser (see Section 3.12). After every command from the ‘Check’ menu, the information in the error browser is updated. If any errors were found, they are displayed here and the error text contains hot-links to the objects that are related to the error (see Figure 15). Either the displayed *objects* themselves contain some invalid attribute values (in this case, click on the object name and correct the attribute value in the object editor), or some *link* to the object does (click on the object to open the object editor, then select the appropriate link editor from there and correct the link attribute).

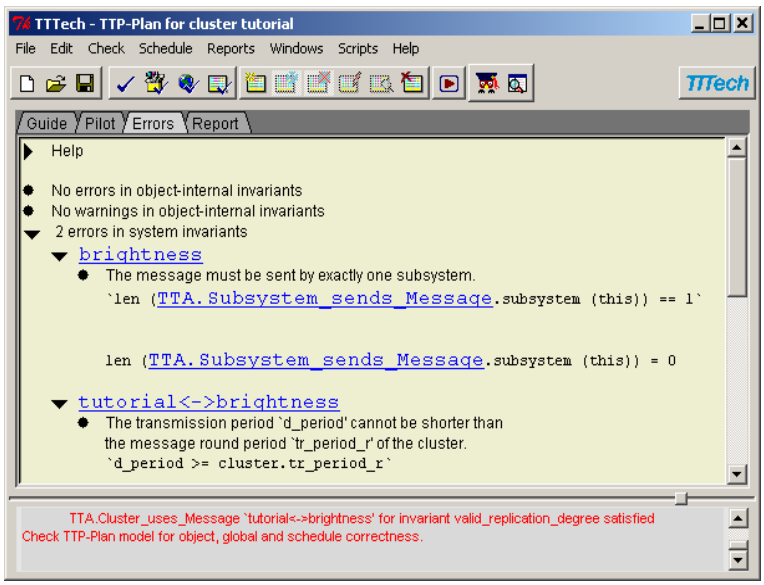


Figure 15: The error browser of TTP-Plan

Another possible error class is constituted by *missing links* (like a message that is not sent by any subsystem) – in this case, click on the unlinked object and open the appropriate link editor from where to create the link.

Schedule errors can also indicate complex error situations that require editing of several objects and/or links to remedy, but the reason for the error is always visible in the error text.

4. The Programming Interface

Besides the interactive interface, TTP-Plan also provides a programming interface which can be used to automate repetitive tasks and to generate reports about a cluster design. It also supports the implementation of interfaces to other tools.

There are several ways to use the programming interface:

- You can start scripts from the ‘Scripts’ menu (see Section 3.2).
- You can run user-specific reports from the ‘Reports’ menu (see Sections 3.2 and 4.6).
- You can specify startup scripts to be run when TTP-Plan starts (see Section 4.7).
- You can execute scripts in batch mode, i.e., run a sequence of scripts in series, using the command line interface of TTP-Plan (see Sections 4.8 and 5).

TTP-Plan’s ‘Scripts’ menu provides the possibility to run scripts interactively. The output of the script is written into the status area of TTP-Plan. The executed script has full access to the cluster design loaded into TTP-Plan and can query or change the objects and links of the cluster.

The programming interface is based on the Python programming language. Knowledge of Python is helpful, but looking at the example scripts in this section and the “Goodies” scripts provided with TTP-Plan (see Appendix B) should be sufficient to let you start using scripts.

4.1. Python Basics

Python is an interpreted, object-oriented programming language. It is easy to learn and accompanied by excellent on-line documentation (see www.python.org). This section gives a short overview of Python’s most important features. For more information, the interested reader is referred to the Python tutorial.[35]

In Python, comments are introduced by #; everything from the # to the end of the line is ignored by the Python interpreter. The end of a line terminates a statement unless the last character is a \ or there is a pending closing parenthesis.

[35] If you have installed Python on a Win32 system, you can access the documentation from the ‘Start’ menu by selecting ’*Program* → *PythonX.Y.* → *PythonManuals*’.

Python is dynamically typed. Therefore, there are no declarations for variables or objects. Attributes of an object are referenced as `object.attribute`, methods of an object are called `object.method (arguments)`.

Python's support for structured data types includes strings, lists, tuples, and dictionaries:

- *Strings* are immutable sequences of characters. Literal strings can be surrounded by either single (`'`) or double (`"`) quotes; if you use `"""` or `'''` as quote characters, the literal string can span multiple lines and will then contain embedded newline characters, e.g.,

  ```
  """This is a string with
     embedded newlines and "embedded" quotes.
  """
  ```

- *Lists* and *tuples* provide mutable and immutable sequences of arbitrary objects, respectively. A literal list is surrounded by brackets, e.g., `[foo, bar, baz]`. A literal tuple is surrounded by parentheses, e.g., `(foo, bar, baz)`. A literal tuple of length one must contain a trailing comma to keep it different from a normal expression: `(foo, )`. Sequence elements are referenced as in `C`: `list [3]` yields the fourth element of the list (because counting starts at 0).
- *Dictionaries* provide hash tables of arbitrary objects. Any immutable object [36] can be used as a key. A literal dictionary is written as:

  ```
  dict = { "foo" : 1, bar : 2.0, baz : [1, 2, 3] }
  ```

  In this example, the string `"foo"` hashes to the integer value `1`, the (immutable) object `bar` hashes to the float value `2.0`, and the object `baz` hashes to the list `[1, 2, 3]`. Referencing of dictionary elements is similar to lists, but the index can be an arbitrary (immutable) object: `dict [bar]` returns `2.0`.

The most unusual aspect of Python is its way of grouping statements: indentation alone groups statements together. Therefore, the statements belonging to an `if`-clause or to the body of a function declaration must be consistently indented relative to the `if`-statement or the function declaration.

Python's control structures comprise `if`, `for`, and `while` statements. Each of these statements is separated by a `:` from its body, and the body must be indented relative to the statement. The following example shows a `for`-statement containing an `if`-statement.

```
# -*- coding: iso-8859-1 -*-
for mu in TTA.Cluster_uses_Message.links () :
    if mu.d_period :
        print mu.message.name, mu.d_period, mu.r_delta
    else :
        print mu.message.name, "no period defined"
```

The `for`-statement of this example iterates over all links of `TTA.Cluster_uses_Message` and prints either the period specified, or the message 'no period defined' for links without a period specification.

[36] Strictly speaking, any object with a hash function that returns a constant value during the whole lifetime of the object.

Python's `for`-statement iterates over the items of a sequence. [37] If you want to iterate over a range of numbers, you can use `for i in range (0, n)`, where `i` runs from 0 to $n-1$.

Functions are defined by the `def`-statement.

For example:

```
# -*- coding: iso-8859-1 -*-
from time import clock
def test (links) :
    t1 = clock ()
    i  = 0
    for l in links :
        if not l.is_defined () :
            i = i + 1
    t2 = clock ()
    print ( "Iteration over %d links needs %5.3f seconds "
          + "(%8.6f seconds per link)"
          ) % (len (links), t2 - t1, (t2- t1) / (len (links) or 1))
    return i

links = Cluster_uses_Message.links ()
n     = test (links)
print ( "%d out of %d links of %s are undefined "
      % (n, len (links), "Cluster_uses_Message")
      )
```

This example counts the number of links of the association `TTA.Cluster_uses_Message` and prints the result on standard output. Besides, it measures and prints the time necessary for the iteration.

The `from...import`-statement loads the module '`time`' and makes the `clock` function visible. If we had used `import time` instead, the call of the `clock` function would have been `time.clock ()`.

4.2. User Interaction

Scripts provide functions for asking the user for values to be used in the script. The four functions `ask_string`, `ask_integer`, `ask_float`, and `ask_list_element` each pop up a dialog window in GUI mode, and let the user enter a value. In batch mode, they expect the exact input on the command line.

Each of these functions requires a `prompt` parameter. The value of `prompt` is displayed besides the entry field of the dialog. The optional `title` parameter sets the title of the dialog window.

`ask_string`, `ask_integer`, and `ask_float` return a value of type `string`, `integer`, or `float`, respectively. Each of these functions accepts the optional `init_val` parameter. The entry field of the dialog is initialized with the value passed for `init_val`.

The function `ask_list_element` uses a list box widget to display a dialog that allows the user choose one element from the `list` passed to this function.

[37] Sequences comprise lists, tuples, strings, and certain user-defined data types.

For instance, a script may ask for one list element and one string in the following way:

```
# -*- Coding: iso-8859-1 -*-
TARGET = ""
while 1:
    source_name = ask_list_element \
        ( prompt = "Subsystem to copy"
        , list   = [ s.name for s in TTA.Subsystem.extension
                        (sort_key = TTA.Subsystem.cmp_key_name)
                   ]
        )
    if not source_name :
        break
    target_name = \
        ask_string (prompt = "Name of new subsystem", init_val = TARGET)
    if not target_name :
        break
    source = TTA.Subsystem.instance (source_name)
    source.copy (target_name)
```

Note: In GUI mode, there are some additional `ask_*` functions which are not available in batch mode: `ask_list_element_combo`, `ask_list_element_spinner`, `ask_open_dir_name`, and `ask_save_dir_name`. For extended support concerning scripting, please contact https://tttech-aerospace.4me.com/.

4.3. Querying the Object Model

This section presents the most important functions for querying the object model of TTP-Plan.

The function `extension` returns all objects of the class it is applied to. For instance, `TTA.Message.extension ()` returns a list of all messages defined for the current cluster. This function supports the inheritance hierarchy of the TTP-Plan object model. For instance, the call `TTA.Frame.extension ()` returns all `TTA.N_Frame`, `TTA.I_Frame` and `TTA.X_Frame` objects.

For associations, you can use the method `links` to get all links of the association. For instance, `TTA.Message_in_N_Frame.links ()` returns all links between N-frames and messages.

For a given object, you can get all links to this object by using the (lowercase) role name of the other partner in the association. For instance, `TTA.Message_in_N_Frame.n_frame (m)` returns all N-frames associated to the message object `m` (i.e., all N-frames containing the message).

Given the class and the name of an object, you can get the object by using the function `instance`. For instance,

```
TTA.Message.instance ("foo")
```


returns the message object named ‘foo’. If the object doesn’t exist yet, `instance` creates it. The function `exists` checks for the existence of an object with a given name.

Given two objects (three in the case of a ternary association), you can get the link between these objects by using the method `link` of the association. For instance, for the cluster ‘`cl`’ and the message ‘`msg`’,

```
TTA.Cluster_uses_Message.link (cl, msg)
```

returns the link between them. You can pass either names or objects to this method.

Given an object or a link, say `foo`, you get the value of an attribute, say `bar`, by the expression `foo.bar`. The function `dir` tells you the names of all attributes of an object or link, for example, `dir (foo)`.

Every root-object class of a tool (i.e., ‘`Cluster`’) has an attribute `.root` which points to the concrete instance, for example, `TTA.Cluster.root`.

If there is only one “main” root object (i.e., ‘`Cluster`’) at any time, `.root` can be skipped and you can access each of the root object’s attributes (but not its methods) in the same way by using the attribute name instead of `root`. These shortcuts can also be used in the entry field of the object editors, e.g., you might enter

```
2 * TTA.Cluster.tr_period_r
```

to specify a period of 2 rounds for the transmission of a message.

To access a specific instance of a root class with *several* instances, however, `.root` has to be used.

One more function that might be useful is `.singleton`. For a class that has only one instance in the tool (e.g., `Cluster`), this function returns this one instance, for example, `TTA.Cluster.singleton` returns `TTA.Cluster 'tutorial'`.

If several instances of a class exist, the function raises an exception. Note that `.singleton` is slighty different from `.root`: Every class that provides `.root` also provides `.singleton`, but not vice versa.

4.4. Changing the Object Model

This section explains how to write scripts for changing the objects or links of a cluster.

To change the value of one or more attributes of an object or link, you use the method `set`. For instance,

```
host_1.set (min_frame_size = 6, max_frame_size = 12)
```

sets the attributes `min_frame_size` and `max_frame_size` of the host `host_1` to the values ‘6’ and ‘12’, respectively.

Generally, `set` expects to get binary attribute values, e.g., an integer value for an integer attribute, a floating point value for a floating point attribute, etc. On the other hand, `set_raw` *always* expects to get string values for attributes, for instance:


```
host_1.set_raw (allow_active_role = "yes")
```

Values entered via the GUI are always raw. Passing raw values might be handy for specifying values for attributes where a complex object is expected. In the database, however, both values (raw and “cooked”) are stored and always synchronized, that means if you change one, the other changes accordingly.

The method `copy` creates a copy of an object with another name. For instance,

```
foo.copy ("bar")
```

returns a copy of the object `foo` with name ‘bar’. The method `rename` changes the name of an object, for example,

```
bar.rename ("baz")
```

changes the name of the object `bar` to ‘baz’.

The method `destroy` removes an object and all its links to other objects from the cluster. If you apply `destroy` to the ‘`Cluster`’ object itself, all objects are deleted.

`define` defines a new object of the specified type and name, for example,

```
TTA.Message.define ("foo", init_value = 0)
```

defines a new message with name ‘foo’ and sets its attribute `init_value` to the value ‘0’. To define a new link, use the method `add` of the appropriate association. For instance, `TTA.Message_uses_Msg_Type.add (foo, bar)` adds a link between the objects `foo` and `bar` to the association. The method `remove` removes a link from an association. For instance, `TTA.Message_uses_Msg_Type.remove ("foo", bar)` will remove the link between the message named `foo` and the message type named `bar`.

The following example demonstrates how to write an on-line script (which you then may run using the ‘Run…’ command from the ‘Scripts’ menu, see Section 3.2):

```
# -*- coding: iso-8859-1 -*-
fmt = "%5d %-50s"
for c in object_types.values () :
    ### don't print 'count' for partial classes
    if (not c.is_partial) or c.count :
        print fmt % (c.count, c.type_name)
for a in assoc_types.values  () :
    print fmt % (len (a.links ()), a.type_name)
```

This script prints the number of objects or links defined for each of the classes and associations of the current cluster design, respectively.

A common use of scripts is the ‘deep copy’ operation, i.e., not only the objects are copied, but also their links to other objects and the linked objects themselves. This script, `subsys_copy.py`, deep-copies a subsystem. That is, it creates a new subsystem `Sub_B`, then generates copies of all the messages the original subsystem sends (naming them after the

original message and appending a _B to the name), generates the links between Sub_B and all the new messages, and duplicates the links TTA.Message_uses_Msg_Type and TTA.Cluster_uses_Message for the new messages. (A new schedule needs to be made after this operation anyway, so no links of TTA.Message_in_N_Frame are duplicated.) After running this script, the new subsystem must only be associated to one or more hosts. Then the new schedule can be built.

```
# -*- coding: iso-8859-1 -*-

def copy_subsys (original_name, copy_name):
    # get the original subsystem
    if not TTA.Subsystem.exists (original_name) :
        return "Cannot find subsystem %s!!" % original_name
    original_subsys = TTA.Subsystem.instance (original_name)

    # make a copy if possible, else forget it
    if TTA.Object.exists (copy_name) :
        return "Cannot create %s - name already exists!" % copy_name
    new_subsys = original_subsys.copy (copy_name)

    print "Deep copy subsystem %s to %s" % (original_name, copy_name)
    for msg_link in TTA.Subsystem_sends_Message.message (original_subsys) :
        # loop over the messages the original_subsys sends
        msg = msg_link.message

        # copy the message
        new_msg = msg.copy (msg.name+"_B")

        # create a link to the new message
        TTA.Subsystem_sends_Message.add (msg_link.copy (new_subsys, new_msg))

        # link the new message to the same c_type the original had
        # note that there can be only one c_type, so no iteration
        # is necessary here
        for link in TTA.Message_uses_Msg_Type.msg_type (msg) :
            msg_type = link.msg_type
            TTA.Message_uses_Msg_Type.add_link (link.copy (new_msg, msg_type))

        # copy all links to cluster to new message
        for link in TTA.Cluster_uses_Message.cluster (msg) :
            cluster = link.cluster
            TTA.Cluster_uses_Message.add_link (link.copy (cluster, new_msg))
    return "Done."
# end def copy_subsys

original_name = ask_list_element \
        ( prompt = "Subsystem to copy"
        , list   = [ s.name for s in TTA.Subsystem.extension
                        (sort_key = TTA.Subsystem.cmp_key_name)
                   ]
        )
if original_name :
    copy_name = ask_string (prompt = "Name of new subsystem", init_val = "SUB_B")
    if copy_name :
        print copy_subsys (original_name, copy_name)
```


Such a script is useful if you want to have two subsystems in a cluster that operate the same way, but are not replicas, like a ‘left hand’ and a ‘right hand’ of a robot, which will have an equal set of messages, but with quite different contents. The way in which the newly created objects (e.g., subsystems and messages) are named is entirely up to the script programmer.

If you need to deep-copy hosts, the script needs to be extended by copying the host object itself, the link `TTA.Host_in_Cluster` for the new host, and the links `TTA.Host_runs_Subsystem_in_Cluster`. You may or may not want to duplicate the subsystems themselves, depending on whether the original and new host are to be replicas (in this case, no duplication of subsystems) or just ‘similar’ hosts with quite different online data.

4.5. Using Dialogs

A script can query the user for object names, attribute values or filenames by using the dialog functions provided by the tools. Each of these dialog functions supports the following parameters:

- `name` specifies a name for the dialog. This argument can be used to uniquely identify the value asked for by the dialog. For scripts, the dialog function will return the `keyword_value`, if a keyword argument with that `name=<value>` was specified in the command line (see Section 5).
- `prompt` will describe the value asked for by the dialog. All dialog functions except the ‘file’ and ‘directory’ dialog functions display the `prompt` value beside the entry field of the dialog.

  If `name` is not specified, the dialog functions will return the `keyword_value`, if a keyword argument `prompt=<value>` was specified in the command line.
- `title` will be used for the title of the dialog window.
- `init_val` will be displayed as initial value in the dialog entry.

  For ‘file’ and ‘directory’ dialog functions, `init_val` will override the values of the arguments `initialfile` and `initialdir`, if both are given.
- `initialfile` specifies the initial value for the basename (all filename dialog functions).
- `initialdir` specifies the initial value for the directory (all filename and directory dialog functions).

The following functions are provided:

- `ask_string` asks for a string and returns the answer provided by the user.
- `ask_integer` asks for an integer value and returns the answer provided by the user.
- `ask_float` asks for a floating point value and returns the answer provided by the user.
- `ask_list_element` asks for one element out of a list of values and returns the answer provided by the user.
- `ask_list_element_combo` is similar to `ask_list_element`, but uses a combo box for the dialog.

- `ask_list_element_spinner` is similar to `ask_list_element`, but uses an entry with spinner buttons and drop-down list for the dialog.
- `ask_open_file_name` asks for the name of an existing file and returns the answer provided by the user.
- `ask_save_file_name` asks for the name of a new file and returns the answer provided by the user.
- `ask_dir_name` asks for the name of a directory and returns the answer provided by the user.

If a dialog function called by a batch script does not find a `keyword_value` corresponding to the argument's `name` (or `prompt`, if `name` is given), it will raise an exception documenting the expected `keyword_value`.

4.6. Report Scripts

A report is a special instance of a hierarchical browser (see Section 3.12). It gives a hierarchical view of some aspects of the cluster design.

A report script is a special kind of script which is activated via the 'Reports' menu.[38] The report script can access the report window with the variable name 'report'.

This 'report' object supports a number of methods, namely:

`clear`, `set_level`, `inc_level`, `dec_level`, `write`, `write_help`, `gauge_activate`, `gauge_deactivate`, `gauge_reset`, `gauge_increment`.

It can be customized by changing the following attributes:

`default_tags`, `header_head`, `header_tail`, `contents_head`, `contents_tail`, `print_node_head`, `print_level_head`, `print_content_head`.

These attributes and methods will be described in the following sections.

The fastest way to learn about report programming might be to look at the report scripts delivered as part of TTP-Plan.

Anatomy of a Report Node

A node of a report comprises up to four parts:

- The 'name' is the first element of the node.
- The 'header' is a (short) text following the 'name'.
- The 'contents' is the main text of the node, which is only shown in the expanded state of the node.
- The sub-nodes are nodes contained in the node. These are invisible when the node is collapsed. When the node is expanded, the 'name' and 'header' of all sub-nodes are shown.

[38] Activating a report script via the 'Scripts' menu will not work!

The ‘name’ and ‘header’ of a node are visible in both the collapsed and the expanded state, whereas ‘contents’ is only displayed in the expanded state. Each of these three parts can be configured via so-called *tags*. A tag is the name for a specific combination of style options. By combining several tags, one can finetune the appearance of a node’s parts (see Section “Tags” below).

The ‘name’ part is required, whereas ‘header’ and ‘contents’ are optional. If defined, a ‘header’ is preceded by the value of the attribute `header_head` and followed by the value of the attribute `header_tail`. The default values for these attributes are a newline and an empty string, respectively.

Similarly, ‘contents’ is preceded by the value of `contents_head` (default: empty) and followed by the value of `contents_tail` (default: empty).

If a node is saved to a file, three more attributes are used:

- The value of `print_node_head` goes ahead of each node (default: ‘`*  `’, i.e., a star followed by three spaces.)
- The value of `print_level_head` is used for each level of indentation in front of a node (default: ‘`.  `’).

  For a node at level 0 (i.e., a top-level node), `print_level_head` is not used. For a node at level *n*, the value of `print_level_head` is inserted *n* times in front of the node’s ‘name’.
- The value of `print_content_head` is used as indentation level for the contents of the node (default: 4 spaces).

  Again, it is inserted *n* times for a node at level *n*.

Tags

A tag is a symbolic name for a set of style options. A tag can control the color, the font or the layout of a report’s element. There may be any number of tags associated with one element. If more than one tag is associated with an element, the style properties defined by the first tag are overriding those of the later tags. If the second tag specifies a style property not specified by the first one, those values are used and so on.

The available tags and their meaning are listed in the following table:

Tag	Meaning
arial	use arial font (this is a variable width font — your mileage may vary)
CR black	use black foreground
CR blue	use blue foreground
CR center	center text
CR courier	use courier font
CR gray	use gray background
CR green	use green background
CR light˙gray	use light gray background
CR nowrap	don't wrap long lines
CR quote	use left and right margin with size of standard indentation
CR red	use red foreground
CR rindent	use right margin with size of standard indentation
CR title	use title font
CR underline	underline text
CR wrap	wrap long lines
CR yellow	use yellow background
CR	

`report.clear`

The method `clear` deletes all nodes (and any other text) from the report. If a report should always start with an empty window, this function should be called at the beginning of the report script. On the other hand, if you would like to accumulate the output of several report script activations, you should not call this function.

Report Levels

The nodes of a report can exist at different levels – a report is a tree-like structure. At any time, the report is at a specific level. Adding another node to the report normally means that the new node exists at the current level.

The 'report' object supports three different ways to change the current report level:

- `set_level` sets the current level to the absolute value passed as parameter. This value must be a number between 0 and 'current level plus 1'.
- `inc_level` and `dec_level` increase or decrease the current level by one, respectively.
- You can pass level-changing parameters to the method `write`, which is used to add nodes to the report. By default, `write` adds the new node at the current level (see Section 4.6 below).

The current level can never be increased by more than one level at a time (because you cannot have a node hierarchy with missing nodes).

report.write

The method write adds a new node to a report. It requires a 'name' parameter and accepts the optional parameters 'header', 'contents', 'open', 'level', 'delta', 'tags', 'name_tags', 'header_tags', and 'content_tags'.

'name', 'header', and 'contents' specify the three parts of a node as described in Section 4.6; 'open' indicates whether the node is initially expanded (the default) or collapsed.

'level' specifies the level at which the node should be added (default: the current level). Alternatively, 'delta' specifies a relative change of level (default: no change).

'tags' specifies tags applying to all 'name', 'header', and 'contents'. The other three '*_tags' parameters apply to their respective node elements. All these tags apply in addition to the report attribute 'default_tags', which specifies tags to apply to all elements of all nodes of the report.

report.write_help

This method adds a node providing a help text about the usage of the report.

Gauge Methods

If the assembling of a particular report takes a long time, you can use a gauge to inform the user about the progress of the work.

You can activate the gauge with the method gauge_activate. This method requires a 'title' parameter which is used as window title for the gauge. The optional 'label' parameter specifies a text displayed above the progress bar. The optional parameters 'g_range' and 'g_delta' specify the range (default: 100) and the default increment (default: 1) for the gauge.

You can increase the gauge with the method gauge_increment, which increases the length of the progress bar by the value of the optional parameter 'delta' (default: 1).

4.7. Startup Scripts

Startup scripts come in two flavors. Pre-startup scripts are run immediately after the start of TTP-Plan, i.e., before any database has been loaded. Pre-startup scripts can be used to customize TTP-Plan. For instance, a startup script can add reports or report categories to the 'Reports' menu. A "normal" startup script runs after the tool has completed the startup, i.e., after loading a database (if any was specified in the command line). For a detailed description of the startup options of TTP-Plan please see Section 5.

A pre-startup script has access to the following functions:

- add_report adds a new report to the specified report category. This function requires two parameters: the first one specifies the name of the report category, the second one the path of the Python script implementing the report.

If the report category does not yet exist, it is automatically created by this function and added to the ‘Reports’ menu.

- `add_report_category` adds all Python scripts found in the specified directory to a report category. This function has one required and two optional parameters. The first parameter specifies the name of the directory and is required. The second parameter specifies the name of the category and defaults to the base-name of the directory. The third parameter can provide a short explanation about the report category and is displayed in the status area (if the report category is selected from the menu).

  If the report category does not yet exist, it is automatically created by this function, and added to the ‘Reports’ menu.
- `add_script_category` adds a script category to the ‘Scripts’ menu and adds all Python scripts found in the specified directory to this category. The parameters of this function are analogous to those of `add_report_category`.

4.8. Batch Mode Scripts

Batch mode scripts are very similar to the scripts run interactively from the ‘Scripts’ menu (see Section 3.2). In fact, every interactive script can be used as batch mode script just as well, as long as it does not use “interactive-only” commands.

Batch mode scripts have access to two additional functions:

- `auto_save` saves the database if it has been changed, or if the program version does not match the version of the database (i.e., if the database had been created by a different program version than the one currently active).
- `keyword_value` returns the value of a keyword argument defined in the command line. The single argument to `keyword_value` specifies the name of the keyword. `keyword_value` returns an empty string for undefined keywords.

4.9. Using Commands

All scripts have access to the commands provided by the interactive interface of TTP-Plan. The function `TTA.Application_Command.run` executes the command specified.

The single argument of `TTA.Application_Command.run` specifies the name of the command to be executed. This name has the form ‘Commandgroup-Name.Command-Name’, e.g., `File.Load` or `Check.Object correctness`. Section 3.2 lists all available commands.

Each command has a precondition, i.e., it can only be executed if that condition is fulfilled. For instance, a database must be loaded to make a schedule. `TTA.Application_Command.run` checks this precondition before executing the command and will raise an exception if the precondition is not satisfied. You can also check the precondition in advance by using `TTA.Application_Command.is_applicable`.

Batch scripts can only execute commands which do not depend on user interaction. For instance, a batch script cannot use any commands from the ‘Help’, ‘Scripts’, or ‘Reports’ menus, except ‘`Help.Bug Info ...`’.

4.10. Automatic Script Generation – the Journal File

TTP-Plan records all user actions in a log file that in fact is a Python script. This file is called `cdbname_username.cdt_journal`, where `cdbname` is the filename of the cluster database and `username` is the login name of the user. By default, this file is located in

`C:\TTP\<toolname>\<version>`.

Try editing a cluster and then take a look at this file. You can run this file or a modified version of it like any Python script from the ‘Scripts’ menu, and all actions recorded will be executed (if possible).

You can use such a journal file as a basis for scripts that automatically generate objects or perform modifications on objects. It can also be extremely helpful if your computer crashed and you have unsaved work, because the actions you performed on the unsaved and lost database are recorded in the journal file and can be repeated by simply running this file as a script.

4.11. Regular Expressions

This section gives a short introduction to regular expression syntax.[39] For a more detailed documentation of regular expressions, the interested reader is referred to the book ‘Mastering Regular Expressions’ by J. Friedl.

A regular expression (or RE) specifies a set of strings that matches it; regular expressions can be concatenated to form new regular expressions; if *A* and *B* are both regular expressions, then *AB* is also a regular expression. If a string *p* matches A and another string *q* matches B, the string *pq* will match AB. Thus, complex expressions can easily be constructed from simpler primitive expressions like the ones described here. For details of the theory and implementation of regular expressions, consult the Friedl book referenced above, or almost any textbook about compiler construction.

A brief explanation of the format of regular expressions follows. For further information and a more pedagogical presentation, please consult the Python Regular Expression HowTo, accessible from http://www.python.org/doc/howto/.

Regular expressions can contain both special and ordinary characters. Most ordinary characters, like ‘`A`’, ‘`a`’, or ‘`0`’, are the simplest regular expressions; they simply match themselves. You can concatenate ordinary characters, so ⌈`last`⌋ matches the string `'last'`. (In the rest of this section, we will write REs in ⌈`this special style`⌋, usually without quotes, and strings to be matched `'in single quotes'`.)

[39] The material in this section is heavily based on the documentation of the “re module” of the standard Python distribution.

Some characters, like '|' or '(', are special. Special characters either stand for classes of ordinary characters, or affect how the regular expressions around them are interpreted.

The special characters are:

'.' (Dot.) In the default mode, this matches any character except a newline. If the DOTALL flag has been specified, this matches any character including a newline.

'^' (Caret.) Matches the start of the string, and in MULTILINE mode also matches immediately after each newline.

'$' Matches the end of the string, and in MULTILINE mode also matches before a newline. ⌜foo⌟ matches both 'foo' and 'foobar', while the regular expression ⌜foo$⌟ matches only 'foo'.

'*' Causes the resulting RE to match 0 or more repetitions of the preceding RE, as many repetitions as are possible. ⌜ab*⌟ will match 'a', 'ab', or 'a' followed by any number of 'b's.

'+' Causes the resulting RE to match 1 or more repetitions of the preceding RE. ⌜ab+⌟ will match 'a' followed by any non-zero number of 'b's; it will not match just 'a'.

'?' Causes the resulting RE to match 0 or 1 repetitions of the preceding RE. ⌜ab?⌟ will match either 'a' or 'ab'.

?, +?, ?? The '', '+', and '?' qualifiers are all *greedy*; they match as much text as possible. Sometimes this behavior is not desired; if the RE ⌜<.*>⌟ is matched against '<H1>title</H1>', it will match the entire string, and not just '<H1>'. Adding '?' after the qualifier makes it perform the match in *non-greedy* or *minimal* fashion; as *few* characters as possible will be matched. Using ⌜<.*?>⌟ in the previous expression will match only '<H1>'.

{*m*,*n* } Causes the resulting RE to match from *m* to *n* repetitions of the preceding RE, attempting to match as many repetitions as possible. For example, ⌜a{3,5 }⌟ will match from 3 to 5 'a' characters. Omitting *n* specifies an infinite upper bound; you cannot omit *m*.

{*m*,*n* }? Causes the resulting RE to match from *m* to *n* repetitions of the preceding RE, attempting to match as *few* repetitions as possible. This is the non-greedy version of the previous qualifier. For example, on the 6-character string 'aaaaaa', ⌜a{3,5 }⌟ will match 5 'a' characters, while ⌜a{3,5 }?⌟ will only match 3 characters.

'\' Either escapes special characters (permitting you to match characters like '*', '?', and so forth), or signals a special sequence; special sequences are discussed below.

If you are not using a raw string to express the pattern, remember that Python also uses the backslash as an escape sequence in string literals; if the escape sequence isn't recognized by Python's parser, the backslash and subsequent character are included in the resulting string. However, if Python would recognize the resulting sequence, the backslash should be repeated twice. This

is complicated and hard to understand, so it's highly recommended that you use raw strings for all but the simplest expressions.

`[]` Used to indicate a set of characters. Characters can be listed individually, or a range of characters can be indicated by giving two characters and separating them by a '-'. Special characters are not active inside sets. For example, ⌜`[akm$]`⌟ will match any of the characters '`a`', '`k`', '`m`', or '`$`'; ⌜`[a-z]`⌟ will match any lowercase letter, and `[a-zA-Z0-9]` matches any letter or digit. Character classes such as `\w` or `\S` (defined below) are also acceptable inside a range. If you want to include a '`]`' or a '`-`' inside a set, precede it with a backslash, or place it as the first character. The pattern ⌜`[]]`⌟ will match `']'`, for example.

You can match the characters not within a range by *complementing* the set. This is indicated by including a '`^`' as the first character of the set; '`^`' elsewhere will simply match the '`^`' character. For example, ⌜`[^5]`⌟ will match any character except '`5`'.

'`|`' The regular expression `A|B`, where A and B can be arbitrary REs, will match either A or B. This can be used inside groups (see below) as well. To match a literal '`|`', use ⌜`\|`⌟, or enclose it inside a character class, as in ⌜`[|]`⌟.

`(...)` Matches whatever regular expression is inside the parentheses, and indicates the start and end of a group; the contents of a group can be retrieved after a match has been performed, and can be matched later in the string with the ⌜`\`*number*⌟ special sequence, described below. To match the literals '`(`' or '`)`', use ⌜`\(`⌟ or ⌜`\)`⌟, or enclose them inside a character class: ⌜`[() ()]`⌟.

`(?...)` This is an extension notation (a '`?`' following a '`(`' is not meaningful otherwise). The first character after the '`?`' determines what the meaning and further syntax of the construct is. Extensions usually do not create a new group; ⌜`(?P<`*name*`>...)`⌟ is the only exception to this rule. Following are the currently supported extensions.

`(?iLmsx)` (One or more letters from the set '`i`', '`L`', '`m`', '`s`', '`x`'.) The group matches the empty string; the letters set the corresponding flags (`re.I`, `re.L`, `re.M`, `re.S`, `re.X`) for the entire regular expression.

`(?:...)` A non-grouping version of regular parentheses. Matches whatever regular expression is inside the parentheses, but the substring matched by the group *cannot* be retrieved after performing a match or referenced later in the pattern.

`(?P<`*name*`>...)` Similar to regular parentheses, but the substring matched by the group is accessible via the symbolic group name *name*. Group names must be valid Python identifiers. A symbolic group is also a numbered group, just as if the group were not named. So the group named 'id' in the example below can also be referenced as the numbered group 1.

For example, if the pattern is ⌜`(?P<id>[a-zA-Z_]\w*)`⌟, the group can be referenced by its name in arguments to methods of match objects, such as `m.group('id')` or `m.end('id')`, and also by name in pattern text (e.g., ⌜`(?P=id)`⌟) and replacement text (e.g., `\g<id>`).

`(?P=`*name*`)` Matches whatever text was matched by the earlier group named *name*.

`(?#...)` A comment; the contents of the parentheses are simply ignored.

`(?=...)` Matches if ⌜`...`⌟ matches next, but does not consume any of the string. This is called a lookahead assertion. For example, ⌜`Isaac (?=Asimov)`⌟ will match `'Isaac '` only if it is followed by `'Asimov'`.

`(?!...)` Matches if ⌜`...`⌟ does not match next. This is a negative look-ahead assertion. For example, ⌜`Isaac (?!Asimov)`⌟ will match `'Isaac '` only if it is *not* followed by `'Asimov'`.

The special sequences consist of '`\`' and a character from the list below. If the ordinary character is not on the list, then the resulting RE will match the second character. For example, ⌜`\$`⌟ matches the character '`$`'.

`\`*number* Matches the contents of the group of the same number. Groups are numbered starting from 1. For example, ⌜`(.+) \1`⌟ matches `'the the'` or `'55 55'`, but not `'the end'` (note the space after the group). This special sequence can only be used to match one of the first 99 groups. If the first digit of *number* is 0, or *number* is 3 octal digits long, it will not be interpreted as a group match, but as the character with octal value *number*. Inside the '`[`' and '`]`' of a character class, all numeric escapes are treated as characters.

`\A` Matches only at the start of the string.

`\b` Matches the empty string, but only at the beginning or end of a word. A word is defined as a sequence of alphanumeric characters, so the end of a word is indicated by whitespace or a non-alphanumeric character. Inside a character range, ⌜`\b`⌟ represents the backspace character, for compatibility with Python's string literals.

`\B` Matches the empty string, but only when it is *not* at the beginning or end of a word.

`\d` Matches any decimal digit; this is equivalent to the set ⌜`[0-9]`⌟.

`\D` Matches any non-digit character; this is equivalent to the set ⌜`[^0-9]`⌟.

`\s` Matches any whitespace character; this is equivalent to the set ⌜`[ \t\n\r\f\v]`⌟.

`\S` Matches any non-whitespace character; this is equivalent to the set ⌜`[^ \t\n\r\f\v]`⌟.

`\w` When the `LOCALE` flag is not specified, matches any alphanumeric character; this is equivalent to the set ⌜`[a-zA-Z0-9_]`⌟. With `LOCALE`, it will match the set ⌜`[0-9_]`⌟ plus whatever characters are defined as letters for the current locale.

`\W` When the `LOCALE` flag is not specified, matches any non-alphanumeric character; this is equivalent to the set ⌜`[^a-zA-Z0-9_]`⌟. With `LOCALE`, it will match any character not in the set ⌜`[0-9_]`⌟, and not defined as a letter for the current locale.

`\Z` Matches only at the end of the string.

`\\` Matches a literal backslash.

5. The Command Line Interface

You can start both the interactive and the batch version of TTP-Plan from the command line (“shell”). In this case, you must either specify the full path of the executable or the path must be part of the `PATH` environment variable.

The executable for the interactive version is named `TTPplan.exe`, the executable for the batch version is named `TTPplan_batch.exe`. See Section 5.1 for details about the batch mode.

Note:

1. `<None>` in the options indicates that there is no default value for the respective option.
2. For GUI-tools, the above options can also be specified prior to starting the tool from the ‘Start’ menu in Windows. Select ‘*Programs* $\rightarrow <$ *company* $> \rightarrow <$ *toolversion* $>$’, right-click on the toolname and select ‘*Properties*’. In the ‘Shortcut’ tab of the dialog window enter the desired option(s) in the ‘Target’ field (`-default_db_dir` is already set during installation). Click ‘OK’, then start the tool as usual.

Example

```
TTPplan.exe -prescripts script1.py,script2.py -commands
"File.Load cluster ..."
```

This sequence will first carry out the scripts 1 and 2, then start the tool and open a dialog window for database selection.

Note:

- The `-commands` option does not take any arguments. For example, it is not possible to enter a filename directly after `"File.Load cluster ..."` If you enter `"File.Load cluster ..." cluster.cdb`, TTP-Plan will treat the first part as a command, and the second part as a database to be loaded *before*, as in the `[db_name]` argument. Hence `cluster.cdb` will be loaded first, *then* the `File.Load cluster ...` dialog window will pop up.

- The `-commands` option also does not allow spaces in command names. If you want to use a command containing spaces, you need to put it in quotation marks or replace the spaces with underscores:
  `"File.Load cluster ..."` or `File.Load_cluster_...`.
- Make sure to use the commands *exactly* as stated in the menu strings. For example, using `"File.Load cluster..."` without a space before the dots will not work!

5.1. Batch Mode

The batch mode of TTP-Plan exhibits a few properties and specifics that should be acknowledged for proper functioning.

- To display help information about the arguments and options of the batch mode, execute it with the option '`-?`' or '`-help`'.
- Many of the menu commands of the GUI are also available in batch mode. See the tables in Section 3.2 for detailed information. The commands with a "yes" in the 'Batch?' column are available here. This information is also provided in the command browser of the 'Help' menu ('command line' is listed in the 'interfacers' of batchable commands).
- In batch mode you have the possibility to write errors to a file to be analyzed later. This is done by redirecting `stdout`.
- In rare cases you may encounter an error message reading "DB Error: The database was changed since you started to work on it . . . ". Mostly this can be fixed by saving the database with another file name as stated in the error message, but sometimes the error can also be caused by an irregular earlier termination of the tool, which left a `.lock` file in the installation directory. Please look up this file and delete it before executing your command sequence again.

  Note: The tool continues execution of the command sequence after this error message, so the result of any step after 'Save . . . ' is unreliable. Please re-run after correcting the error.

Keyword Arguments

For the batch version, you can also specify *keyword arguments*. A keyword argument is specified using `key=value`. In scripts they can be used via the function `keyword_value`.

For instance:

```
TTPplan_batch.exe -commands "Help.Bug info" bug_report=bug_info
```

will create a file `bug_info.txt` with the requested information in the installation directory.

For details see Section 4.5.

Note that, to get the complete information in the bug report, you need to add the `"Help.Bug info"` command at the end of the command sequence that caused the error, and repeat the

execution of the tool. If you execute the "`Help.Bug info`" command alone, only product and system information will be written to the report.

Similar to the `-commands` option, spaces (and some special characters) are not allowed in key values. If you want to use such a value, you need to put it in double quotes.

6. Scheduling

6.1. Cluster Scheduling and the Fixed Round Number (FRN)

The cluster schedule is calculated by the scheduler on the basis of various user inputs. The schedule defines important properties of the cluster, such as the number of rounds in the cluster cycle, the duration of rounds, and at what points in the cluster cycle different messages are sent.

The number of rounds in a cluster schedule is a key property and is determined in either of two ways. In the first case, the optimum number of rounds is computed by the scheduler based on two user-specified attributes: the round duration (`tr_period` of `Cluster`) and the message period (`d_period` of `TTA.Cluster_uses_Message`). This value (number of rounds) is generated automatically along with other cluster schedule properties.

In the second case, a fixed number of rounds is specified by the user – the *Fixed Round Number (FRN)* (see `Cluster` attribute `fixed_round_number`). The FRN constrains the optimality of the resulting cluster schedule, but is useful for interfacing with other clusters. (For example, if you need to interface with a system that has 10 rounds, you can specify 10 rounds as the FRN of the system you are designing.)

The user-specified `tr_period` (round duration) and `d_period` (message period) attributes are crucial to the computation of the cluster schedule. The `d_period` has a definite relationship to the `tr_period`. Normally, the `d_period` is a value that is a power-of-2 multiple of the `tr_period`. For example, if the `tr_period` is 1000 microseconds, then the `d_period` could have a value of 2000, 4000, 8000, or 16000 microseconds, and so on. If the user specifies a non-power-of-two value, then the nearest lower power-of-two value is used by the scheduler (for example, 2000 microseconds if 2500 microseconds has been specified). The power-of-two computation will always provide efficient cluster scheduling since any number of messages can easily be accommodated within a cycle even while maintaining the efficiency of rounds (i.e., the bandwidth).

With the two cluster attributes `min_tdma_rounds` and `max_tdma_rounds` you can also influence the scheduler, without restraining it as much as with `fixed_round_number`. This can, for example, be useful if you intend to monitor your cluster with TTP-View later, because TTP-View requires a minimum round number of 2.

When an FRN value has been specified, the following relationship between `d_period` and `tr_period` exists.

```
d_period = tr_period * divisor of FRN
```

For example, when the FRN is specified as 10 rounds, then only the following `d_period` values are possible:

```
d_period = tr_period * 1
d_period = tr_period * 2
d_period = tr_period * 5
d_period = tr_period * 10
```

Clearly, in a cycle in which more than one message has to be transmitted with the `d_periods` in the above example, the probability of two or more messages occurring in the same round increases greatly. When this happens, the duration of the round will be increased, leading to wasted bandwidth in other rounds.

Note: making a schedule erases the contents of the `.ddb` directory, so please do NOT save any files to this directory!

6.2. Incremental Scheduling – Schedule Extension

Incremental scheduling (a.k.a. *schedule extension*) means that scheduling is done in discrete steps. Each time you make a schedule, this is called a "schedule step". These schedule "steps" do not really form a sequence of *different* steps, but the whole process is a more iterative procedure: you change something and make a schedule, then look at it, change another thing and make a schedule, look at it, and so on. Each such cycle of changing and scheduling is considered a *schedule step* (they are also referred to in the description of the 'Schedule'menu in Section 3.2). You can make as many schedule steps as you like, until you are satisfied with the result. *Note:* only "frozen" schedule steps are stored and actually *counted* as steps (schedule steps are numbered to identify them later on).

You can keep a schedule (by "freezing" the current schedule step) and add new messages[40] to it, thus filling the "holes" in the original schedule without changing it. Other possible additions are:

- additional hosts and subsystems,
- additional message types,
- mapping of new subsystems to hosts

The first schedule step (also called the 'base step') contains all information necessary to make the MEDL for each host. In later schedule steps, additional messages can be added for transmission in previously unused portions of frames. Since the MEDL only

[40] with their type, period and sending subsystem!

contains information about the lengths of the frames, but not their contents, the addition of messages can be done without changing the MEDL.

If, for example, only two hosts `A` and `B` need additional messages, only these two must be updated, while all other hosts can remain at the base step of the scheduling. Later, host `C` might be updated to use the second schedule step, too. Eventually, hosts `A`, `D`, and `E` might get updated to yet another schedule step with additional messages. At runtime, a cluster using incremental scheduling can thus contain hosts with differing schedule steps.

The `Cluster` attribute `allow_schedule_extension` specifies whether to enable incremental scheduling or not. If it is set to 'yes', the command 'Schedule.Freeze current schedule' freezes the current schedule step, i.e., the commands 'Schedule.Make new schedule' and 'Schedule.Delete schedule' leave the frozen schedule steps unchanged and add/remove a new/unfrozen schedule step. The command 'Schedule.Thaw frozen schedule step' allows to "unfreeze" the most recently frozen schedule step. This means you can continue editing the schedule where you left it before freezing, your previous modifications are still there.

For the `Host` object of TTP-Plan there is also an attribute `allow_schedule_extension`. As long as a host has this attribute set to 'yes', the scheduler will include that host in new schedule steps. For `allow_schedule_extension` to make sense, the host needs a `min_frame_size` larger than needed for base step. A generous `min_frame_size` reserves bandwidth for additional schedule steps.

When a host has exhausted the spare capacity of its frames, or is known not to want to participate in any further schedule steps, its attribute `allow_schedule_extension` must be set to 'no'.

To allow for safe interoperation of hosts at various steps of an incremental schedule, each of the hosts participating in a schedule step sends one message per schedule step carrying the schedule-step checksum (computed by the TTP-Plan scheduler) which allows on-line consistency checks. For a schedule step to be safely usable, the schedule-step checksum sent by the sender must be equal to the schedule-step checksum expected by the receiver. The cluster attributes `schedule_step_cs_bytes` and `schedule_step_cs_period` define the size and period of the schedule-step checksum messages used in the cluster.[41]

6.3. Multiplexing and `MUX_Ghosts`

Multiplexing means that two or more hosts use the same sending slot in different rounds. This can be specified in the `Host_uses_Slot` attributes `mux_period` (period of multiplexing) and `mux_round` (first round when this host sends). By default, both attributes are set to 1 which switches off multiplexing: the host sends exclusively in a slot of its own.

In this context a special kind of host has been designed to be non-periodic and still participate in the multiplexing. (*Note:* the *message* of this host is still periodic!) It meets additional requirements like the following:

[41] Schedule-step checksums can be omitted (by setting `schedule_step_cs_bytes` to 0), but this is not recommended.

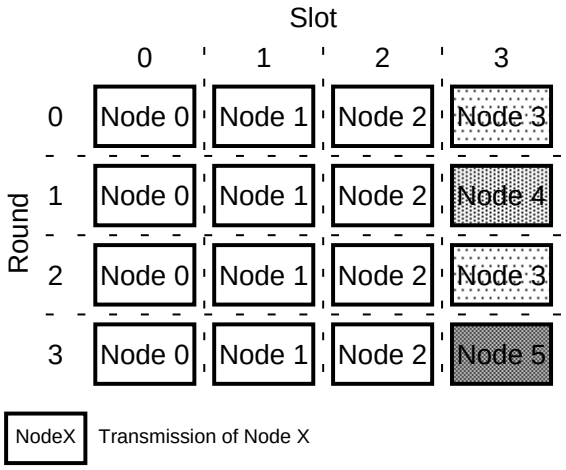


Figure 16: Multiplexed slots

Example scenario:

- One slot (in a schedule of 32 rounds) shall be shared by 6 controllers.
- Each controller shall be assigned one round-slot every 8th round (periodic data).
- In the remaining 4*2 rounds (2 per multiplexing period), each controller shall be assigned one additional round-slot (event data, higher-level protocols).
- With controllers A to F, the 32 round-slots shall be shared like this:

```
|A B C D E F|A B|A B C D E F|C D|A B C D E F|E F|A B C D E F|? ?|
```

- The remaining two round-slots (marked "??") can be assigned to any multiplexing partner.

The pattern required is non-periodic in the sense that transmissions by one multiplexing host are not separated by a constant number of rounds anymore. However, it can still be modeled by assigning *multiple* periods to a single *multiplexing* host (e.g., in the above example, both `mux_periods` 8 and 32 could be assigned to the same host).

This type of host is called `MUX_Ghost` (in the following simply called "ghost") and has the following properties:

- It behaves like a host in that it can run subsystems in a cluster,[42] and can thus send messages. In addition, it must be assigned a `mux_period` and a `mux_round` (in `Host_uses_Slot`).
- It is linked – via a `Host_has_MUX_Ghost` link – to a specific host which implements the subsystems specified for the ghost. (*Note:* a ghost must be linked to the same slot as the linked host.)

[42] See the corresponding association `TTA.Host_runs_Subsystem_in_Cluster`.

- A ghost has no `Host_in_Cluster` link.
- A ghost has no MEDL.
- The MEDL of a host contains the hosts own `R_Slots` and the `R_Slots` of all ghosts linked to it.

Ghosts are created by single-clicking on the 'Host' object in the 'Pilot' and selecting `MUX_Ghost` from the menu. You link a ghost to a host by selecting the `Host_has_MUX_Ghost` link from the 'Edit.Links' menu, and to the same slot as its host via `Host_uses_Slot` (there you also set the attributes `mux_period` and `mux_round`). The scheduler then links R-slots and frames to the "ghost-master" (i.e., the node that has the MEDL for the ghost).[43]

[43] TTP-Build will consider all subsystems, R-slots, messages, and frames of ghosts as if they belonged to the ghost-master.

7. The Replica-Deterministic Agreements (RDAs) of TTP-Plan

Replica determinism is important whenever a software component is replicated, i.e., when there is more than a single instance (replica) of a component in a system. Replication is used to increase the fault tolerance of a system. If one of the replicas fails, the other replicas can still provide the functionality of the replicated component. For a system to be correct, all replicas of a component must behave identically (replica-deterministic). That means that:

- the replicas must receive the same input
- they must deliver the same output
- they must do it at the same time
- the underlying processes must make the same decisions

One source of replica nondeterminism are readings from replicated sensors. To ensure replica-determinism in the presence of replica-nondeterministic message values, it is necessary to perform a replica-deterministic agreement (*RDA*). An RDA takes the raw values of all living replicas and computes a replica-deterministic result.

TTP-Plan provides a number of predefined replica-deterministic agreement algorithms (the complete list can be found in the optional attribute `agreement` of `TTA.Msg_Type`):

1. `RD_1_valid`: this is the default agreement if nothing else is specified or if the message is non-replicated.[44] It uses one of the valid raw values; suitable when all raw values are known to be identical.
2. `RD_1_valid_strict`: compares all valid raw values and only uses the (common) value if they are all the same.
3. `RD_Add`: returns the sum of valid raw values.
4. `RD_Average`: returns the average of the valid raw values. *Note:* if the sum of the raw values during calculation exceeds the allowed range of the data type, the result may be wrong! Hence this agreement is only to be used if you know your value range very well and want a fast algorithm.

[44] In the latter case `RD_1_valid` is *always* used; any user settings are ignored.

5. `RD_Average_Full_Range`: similar to `RD_Average`, but without the risk of overflow. In some cases[45] slower, but safer than `RD_Average`.
   Explanation: if there is a risk of overflow, TTP-Build first tries to use the next larger data type for the sum of values to avoid the problem. If there is no larger data type available (because your compiler does not support it), an algorithm is used that does not produce overflows, but is somewhat slower. The standard `RD_Average` algorithm cannot deal with this kind of problem.
6. `RD_M_Vote`: "majority-vote", selects the raw value that has the greatest number of occurrences. Note that this algorithm calculates an *absolute* majority (e.g., 4 out of 7) and hence works only on an uneven number of replicated subsystems (in order to avoid decision problems).

Note: agreements can only be calculated for atomic data types; for structured messages (arrays and structs), only the raw values are accessible, hence only `RD_1_valid` can be used.

For some examples for the use of the different agreement types refer to the TTP-Build manual, Section "Replica-Deterministic Agreement (RDA) Function – Failure Strategies".

Hint concerning floating point values in agreements

Some of the above algorithms, `RD_M_Vote` and `RD_1_valid_strict`, compare the values of the replicas by strict *equality* ("=="). This can cause severe problems when using floating point values because the representation of such values is strongly hardware dependent (CPU, compiler, ...). For example, if the value is supposed to be 3.000...0, it may come out 2.999...9 on one target and 3.000...1 on the other one. In strict comparison, these two can never be equal, hence the above agreements cannot be calculated. So if you want to use one of these agreements on floating point values, make sure that you really have *identical* components in your system or use a different agreement.

[45] e.g., if you are using 16-bit data types and your compiler does not support 32-bit ones.

8. H-State and Reintegration

8.1. The H-State

Replication of subsystems clearly sounds like a great idea to achieve fault tolerance, and if nodes start up at the same time and were correctly designed to behave in a replica-deterministic way, that should give us the desired fault tolerance: if one of the subsystem replicas fails, the ones still active will continue to provide the necessary services/messages to the system.

A node failure can be permanent; in this case, the failed replica will just stay inactive until a system repair is performed. But often a failure is only temporary, like a short power surge or a flipped bit in the RAM of the node. Ideally, we want subsystems on a node that temporarily fails to fully recover – meaning they should perform a restart, followed by some self-tests, and if no permanent faults are detected, the subsystem should become a replica again, and therefore behave as if the node had never failed.

The functionality that a subsystem offers is part of the global system design. When it comes to implementing a subsystem, the overall subsystem functionality is split into *tasks*. A common term when talking about operating systems, tasks are functions without parameters or return values, which communicate through messages.[46] Each task generally takes some input, performs some function on it, and produces a result as output, where input and output are messages. Furthermore, the task can contain static internal data that influence the computation and hence the output. This can be formulated as follows: a task that performs the function F on *input* and produces *output* contains internal data called *h-state*,[47] so that the output depends on both the input and its current h-state: $output = F(input, h-state)$.

In the course of this calculation, the h-state changes continuously as well, resulting in a dynamic behavior of F (i.e., the same input will result in a different output at different times).

[46] Blocking mechanisms between tasks, like semaphores, are not used in a strictly time-triggered system.

[47] The 'h' stands for 'history', meaning that this piece of data is usually cleared at system startup and is updated in the course of system operation.

Examples:

- A trivial function that outputs a constant value has an empty input and an empty h-state.
- A function that takes two numbers as input and outputs their sum has an empty h-state, since the sum of the two inputs does not require any additional internal data.
- A function that takes a number as input and returns the sum of all numbers since system startup (an integrator) has an h-state item which could be called `sum_so_far`; this h-state is initialized to zero at startup and needs to be remembered between any two calls of the function.
- A function that gets a raw sensor reading as input and delivers an average over the last five readings as a result – a *smoothing* function for the sensor readings – needs to store the last four readings in a circular buffer which constitutes the h-state; when the function is called, the input is used as the fifth value for the average, resulting in an output; then the oldest of the four values in the buffer is replaced by the new value, and thus the h-state is updated.[48]

  Filter functions like smoothing generally have h-states.

Assume the smoothing function in the last example above were to be implemented in a replica-deterministic fashion, so that we can have two (or more) replicas running in a system. When the replica nodes start up (at the same time!), they initialize their buffers, and while they are all running, they will produce identical output, if they receive the same input (to ensure this is the task of the communication subsystem).

If one of the replicas fails temporarily and restarts, it can only become a useful member of the replicated group again if it can get hold of the current buffer contents of the running replicas – i.e., of their h-state – and hence becomes aware of the system's previous history. This requires that the h-state is transmitted by an active replica so that it can be picked up by a restarting node. [49]

The phase during which a subsystem picks up its h-state from other, active replicas, is called *reintegration* and is described in the following subsection.

8.2. Reintegration

When there is a node restart, the controller checks the current state (or mode) of the cluster. If it is in Startup Mode, the restart is considered a fresh startup. If the cluster is in "working" mode, then the restart is considered a reintegration. (*Note:* the reintegration concept pertains to subsystems, as opposed to nodes!)

[48] Depending on the implementation, such a function could also have a counter telling how many values in the buffer are already valid, which can be important during the startup phase when less than four values are present. This is important to know, because that counter would be part of the h-state as well.

[49] In this example, a 'slow restart' is also possible: the restarting node waits for five calls before answering, filling its internal buffer. But often this is not possible, e.g., if the values in the buffer depend on the previous buffer values, and it can take longer than is acceptable to integrate in this way.

After falling out of a TTP system, a subsystem can be set to either reintegrate into the system or not. There are different reintegration strategies, which can be selected in the `reintegration_type` attribute of the `TTA.Subsystem` object defined in TTP-Plan. If this attribute is not specified, then the particular subsystem will not reintegrate. Currently, the following strategies are available:

- No h-state, no reintegration (`No_Reintegration`).
  This means, once the subsystem has failed, it will not recover its functionality, i.e., its tasks will not run.
- Reintegration without h-state (`Reinit_Reintegration`).
  If no h-state is present, it must be initialized. This is the case during system startup (i.e., when no node is yet active), and also for independent subsystems, (e.g., those reading their own sensors and not getting information from other subsystems). [50] To be able to reintegrate without an h-state, all data items must have an `init value` that is used when no 'current' data is yet present. These values are typically selected to present 'safe values', like a ticket number starting at zero. If there is no safe value for a message, the message must be marked as unavailable until it can be updated properly. In this case, the `init value` of the message is ignored by the receivers.
  To allow for a task to start with an `init value`, it is necessary to set the optional attribute `may_run_with_init_value` of the `App_Task` object, as described in the "Optional Attributes" section of the TTP-Build manual.
- Reintegration with h-state (`H_State_Reintegration`). For the h-state to be used in reintegration, it needs to be visible from "outside" its subsystem as well (a subsystem wanting to reintegrate cannot look inside another subsystem for some internal data). Therefore the user has to define a global message ("h-state message") that contains this information. This is done in the attribute `is_h_state` of the `TTA.Message` object in TTP-Plan. Now the output can be considered solely a function of the input, no "hidden" data is involved anymore. These h-state messages are only received on the bus when *no* valid h-state is currently present.
  In the TTP-Tools so called *task guards* are used to ensure that a reintegrating subsystem gets the complete h-state information from all other subsystems. They work similar to pre-task-hooks (described in the TTP-Build manual, section "Classes"), i.e., TTP-Build checks whether all necessary h-states for a task are present. This means that either
  - the subsystem is already integrated, or
  - all h-state messages were received and valid.

  The time used for this check is predefined in TTP-Plan in the (view-only) attribute `guard_time_budget` of the reintegration types in the `TTA.Subsystem` object, and is automatically added to the total time budget of the task for scheduling.
  The time needed for reintegration is application-specific; after identifying all h-state items on a subsystem, the duration of reintegration on (regularly retransmitted) h-

[50] This can also occur when all subsystem replicas happen to fail at the same time during normal system operation; however, this probably indicates that the fault hypothesis for the system needs to be reevaluated.

states from replicas can be determined from the cluster schedule. This value can be considered the “best case”, i.e., if all h-states are received correctly.

If no valid h-state is received, a special counter is incremented for the current round. When the number of such “invalid” rounds exceeds a certain threshold, the subsystem gives up waiting and performs a coldstart based on its `init values`, as in the `Reinit_Reintegration` scenario. This threshold is specified by the user in the `reintegration_timeout` attribute of `TTA.Subsystem`. It can be different for different application modes of a node and is usually given in integer multiples of the cluster cycle.

The advantage of this procedure is that there *always* is a chance for reintegration, even in the unlikely case that all replicas of a subsystem fail simultaneously. The `reintegration_timeout` presents the worst case scenario; at this point, at the latest, reintegration is initiated.

An attribute that *must* be set for h-state reintegration is the *subsystem status* (i.e., the optional attribute `status` of the `TTA.Subsystem` object in TTP-Plan), else TTP-Build will raise an error. If the `status` is set to ‘yes’, the subsystem will reintegrate as soon as its own h-state is complete, and not wait for information from the others. A message from a subsystem is hence only considered valid if both receiver status, sender status *and* subsystem status are valid. This is to prevent faulty subsystems from “infecting” others. If the status is set to ‘no’, the subsystem will depend on the h-states of the other subsystems and only reintegrate when all of them have been received.

The attribute `use_full_h_state_as_guard` of the `TTA.Subsystem` object can be set by the user to ensure that all tasks wait for the whole subsystem (messages and h-states) to be checked before they start. Otherwise the first task can start directly when it has received its own h-state and does not have to wait for the rest.

9. Remote Pin Voting (RPV)

The time-triggered architecture relies on fail-silence as an important safety aspect of node design. However, in applications where nodes control *external* hardware, you cannot assume that the node will always fail in such a way that it also safely shuts down the external hardware.[51] Therefore a mechanism has been implemented to allow for distributed checking and "remote removal" of a faulty node.

The *Remote Pin Voting* algorithm (RPV) is an FT-COM functionality that enables a communication controller to control host output for external devices in order to shut them down safely in case of failure. This means that a group of "healthy" nodes within the system can bring a faulty node into a safe state without that node's cooperation. A major advantage of this mechanism is that the system can *independently* detect a faulty CPU, even if The CPU itself signals that it is healthy. In this respect, the controller has indirect control over the host application, thus saving an extra "FT-COM CPU" besides the host CPU.

However, what happens if such a node that has the power of remote removal over another node becomes faulty itself and requests a *healthy* node to be removed? This "error propagation" must be prevented by design if it poses a threat to the availability of the whole system!

The method to remove a node should:

- be performed over the common communication bus
- allow fault-tolerant operation
- allow application specific strategies (i.e., configuration)

To demonstrate how such a mechanism would work, let us consider a system consisting of four nodes (A, B, C, D). Nodes A and C control the same actuator. By default, A controls the actuator, and C only takes over when A is deemed faulty and thus removed from the system. However, since it is possible that A does not register itself as faulty, you cannot rely on a fault-detection mechanism only originating from A. Instead, it is better to let other nodes in the system check on A independently, in order to ensure that a fault in A is detected correctly.

[51] Imagine a node controlling an actuator fails silently and leaves the actuator running at full speed!

We denote the set of nodes that participate in the voting algorithm of node A with `voters(A)`, for example `voters(A) = B, D`. Only these nodes may send voting messages with respect to the voting function of node A. Only node A receives the voting messages and performs the voting algorithm.

(Of course, the same algorithm can be performed for another node in parallel; e.g., node B. In this case we would also have a set `voters(B)` that is most likely unequal to the set `voters(A)`.)

The `voters(A)`, B and D, could check on A in the following way: The system design arranges for B and D to get the same input as A, make the same computations, and then check the individual results against the output of A. If, say, the difference between the result computed by B and the output of A is greater than a certain limit, the message sent by B will appropriately reflect this. (Node D behaves accordingly.)

This brings us to the messages that indicate a faulty node. These messages are designed to *indicate* the condition of A, and also to effect a suitable *action* depending on A's condition.

B and D send special messages as part of their frames that are used for the voting algorithm on A (the message type is a single bit, i.e., the message can have a value of either `0` or `1`). The controller of A interprets the messages from B and D (i.e., the "Remote Pin Votes") and increments a local counter accordingly. At configuration specific intervals (no longer than one cluster cycle, no shorter than one round), the controller of A has a *decision point*: It compares the counter with a *threshold* (given in its MEDL) to determine if a special pin needs to be asserted.[52] If the result exceeds the threshold, this pin would typically be used by appropriate hardware mechanisms to remove power from the local actuator, thus bringing it to a safe state. Then the counter is initialized with zero again.

RPV messages can be sent only once per round, but several times in one cycle. The counter on the node that is being checked can be evaluated several times in a cycle. This offers considerable flexibility for checking the state of the node.

It is important to note that, since the RPV counter threshold is an absolute value, the failure of an RPV node could result in the threshold not being exceeded. This is particularly crucial if the action to be taken upon threshold crossing is to stop the node.

RPV parameters

The parameters for Remote Pin Voting can be set in TTP-Plan for the group of RPV nodes, known as the `RPV.Group`.[53] The required parameters are listed below (and described in more detail in Section 12.1, "RPV.Group: required attributes"). For the optional attributes please refer to Section 12.2.

- `decision_point`: this specifies the points (round-slots) at which the RPV counter will be evaluated (i.e., the value of the external pin is updated). Default is "(-1)

[52] The threshold (specified in the attribute `threshold`) can be reached from below or from above. In the first case the pin is tallied when *enough* votes have been counted. In the other case the pin is left untouched as long as the counter does not go *below* the threshold.

[53] Some can also be set in TTP-Matlink.

every 1 from 1", which states the last round-slot in the cluster cycle. For a detailed explanation see the syntax description in "RPV.Group: required attributes".

- `init_value`: value used by controlling hosts immediately after startup. Can be 0 (no) or 1 (yes).
- `r_period`: period of associated RPV message (in rounds).
- `threshold`: sets the threshold value of the RPV counter. If this value is exceeded[54], the action defined in `TTA.Host.rpv_startup_logic` (see note below) is taken.
- `threshold_bias`: after a positive RPV decision (counter > threshold), this value is added to the number of RPV messages (sent with the value "1") received for the evaluation at the next decision point.
- `voting_logic`: if this parameter is set to "yes", the RPV pin defaults to high and is set to low when the RPV logic trips (i.e., when the number of RPV messages sent with value "1" exceeds the threshold).

The optional attribute `Host.rpv_startup_logic` indicates whether the RPV pin should initially be set to high (yes) or low (no). If nothing is specified, the default value 'no' is used.

Considerations regarding voting logic

With *negative logic* the messages are interpreted as "kill" messages: As long as the number of messages within a voting interval stays below the threshold, the pin is not asserted. If the number of messages exceeds the threshold the pin is asserted. Incorrect or missing transmissions therefore favor the aliveness of the receiver and thus higher availability.

With *positive logic* the messages are interpreted as "healthy" messages: As long as the number of messages within a voting interval stays below the threshold, the pin is asserted. Incorrect or missing transmissions therefore favor the shutdown of the receiver and thus higher safety.

The main difference between negative and positive logic is the resulting availability of the node. With negative logic a node A can still be active, even when all nodes in `voters(A)` have failed. On the other hand, with positive logic a minimum number (depending on the threshold) of active nodes in `voters(A)` are needed. Otherwise the node would not get enough "healthy" messages and the pin would be asserted.

[54] i.e., by the number of voting messages (+ `threshold_bias`) with value "1" received prior to the decision point

10. The Monitor MEDL

The Monitor MEDL is a special, node-level MEDL that is generated automatically by TTP-Plan after scheduling. This MEDL is loaded into the communication controller of the Monitoring Node that is used by the monitoring tool TTP-View. This MEDL contains a special CNI message area layout that is required by the host software operating within the Monitoring Node.

The node-level information of the Monitor MEDL does not interfere with node-level designs of the cluster; however, changes to the cluster design render the Monitor MEDL invalid.

In multiweb clusters, the Monitoring Node has to be configured individually for each web, hence Monitor MEDLs can only be generated for one web at a time. This web is specified in the cluster attribute `active_web`. After scheduling, the generated download database (DDB) contains all necessary information for download and monitoring of this web. *Note:* to monitor another web, you only need to change the selection of `active_web` accordingly and schedule again. As long as you do not change anything else, the schedule checksum remains unaltered, so you do not need to reschedule all nodes, you can directly download the new Monitor MEDL and start monitoring.

11. The Object Model

The design of a cluster specifies the objects comprising the cluster and their temporal behavior. Each of these objects is characterized by a number of attributes and by its relations to the other objects of the cluster.

The essential parts of the TTP-Plan object model are shown in Figure 17. Object classes are displayed as rectangles, associations between object classes as connecting lines. Read-only classes and associations are displayed in light gray, classes and associations you can change are displayed in dark gray.

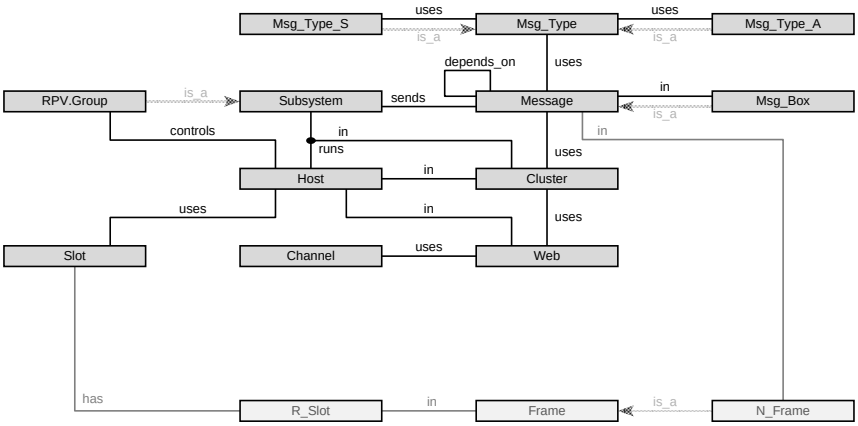


Figure 17: The essential object model of TTP-Plan

The object model of TTP-Plan is structured in a way that minimizes the coupling of the objects of a distributed system. Low coupling is the cornerstone for the achievement of composability, reusability, testability, and maintainability.

A cluster design specifies the properties of a distributed real-time system. These properties act as constraints on the software systems implemented on the individual nodes of the distributed system.

A cluster comprises a number of nodes (organized into one or more *webs*, see description in Section 11.1) that are connected via a bus.Each cluster has its own specific pattern for the transmission timing of the messages sent by the hosts of the nodes.

Each node is *synchronized* to at least one web in one cluster,[55] except if it acts as a *gateway* between clusters. In this case the node needs to have two controllers, each synchronized to one of the clusters.

In a cluster, each of the hosts executes a number of *subsystems*. The subsystem is the basic unit of packaging software components; as such, it is the unit of distribution, replication, and composability; also the fault-tolerance and error-detection strategies are applied here. At the cluster level, a subsystem is a black box without attributes – its only role is to serve as a unit of distribution/replication and for packaging messages.

Messages are used for the communication between different hosts of the cluster. They have state semantics (hence *state messages*), which means they contain information about the current *state* of system parameters (like 'current temperature' or 'current pressure'). Each message has a specific message type (`TTA.Msg_Type`). In contrast to the state messages, *event messages* carry information about *changes* of state (for example, 'temperature has gone up one degree'). They are not used in strictly time-triggered systems.

The cluster design should not contain any information about the design of the individual nodes of the distributed system. Therefore, the cluster design must specify the messages *sent* by the various hosts (because this is globally relevant) but it should not prescribe the messages *received* by a host (because that is a local design decision for each host).

11.1. Classes

This section presents all classes of TTP-Plan's object model. A more detailed description of the attributes of these classes is given in Sections 12 to 12.3.

`TTA.Channel`

This class contains the bus channels of the cluster, and allows you to add a description to them. Each instance of the class describes one channel. By default there are two channels per web (limited by the communication controller), and each channel is automatically linked to the current web (`TTA.Web_uses_Channel`) and – via `TTA.Cluster_uses_Web` – to the cluster. For single-channel operation of a web, one channel has to be removed manually.

[55] Synchronization to multiple webs is only possible if the webs are synchronized to each other.

TTA.Cluster

The class TTA.Cluster defines all attributes of a cluster. These attributes belong to the following categories:

- *Communication:* the communication attributes define some global parameters of the communication between the hosts of the cluster. They comprise the duration of the round (tr_period), constraints on the number of rounds (min_tdma_rounds, max_tdma_rounds) and on the size of the frames transmitted (max_frame_size), and some others.
- *Hardware:* the hardware parameter controller_type defines the types of communication controllers used in the cluster.
- *Identification:* these parameters describe the application running on the cluster, e.g., the ID and version of the application (application_id, application_version).

The associations of the class TTA.Cluster define the hosts building the cluster and the modes the cluster operates in. In TTP-Plan there can only be one TTA.Cluster object at any given time.

TTA.Frame (view-only)

The class Frame actually denotes the "superclass" of all different frame types, i.e., I-frames, N-frames and X-frames (see respective descriptions). This is also reflected in the associations I_Frame_is_a_Frame and N_Frame_is_a_Frame.

TTA.Host

The class TTA.Host defines the attributes of a host in a cluster. These attributes belong to the following categories:

- *Communication:* the communication attributes define the communication parameters of the host. They include constraints on the size of frames sent by the host (max_frame_size, min_frame_size), which are then used as input data by the scheduler. Note that the max_frame_size may also be increased if necessary, the default value does not represent the absolute maximum.

  In addition, the timing of the user interrupts raised by the communication controller can be specified.
- *Hardware:* the hardware parameters define, for example, the minimum propagation delay incurred by the bus driver of the host, or the base address of the host's CNI.

  Note to users of Manchester encoding: due to a bug in the C2NF (AS8202NF) controller, you have to set the attribute asynchronous_preamble_cutoff_ns (of TTP.Controller_T3C2NF) with care. Please refer to the TTTech Chip IP application notes AN133, AN134 and AN136 for details (available on the TTTech homepage, http://www.tttech.com/products/ttp/asics-ip-models/application-notes/).

- *Identification:* these parameters identify the host for TTP-Load.

The associations of the class `TTA.Host` define the subsystems executed by a host and the sending slot used by a host, as well as which web the host belongs to. *Note:* some attributes of this class and its associations are defined in TTP-Build and hence described in the TTP-Build user manual.

`TTA.I_Frame` (view-only)

The class `TTA.I_Frame` defines the attributes of an I-frame transmitted on the bus. These attributes include the address of the frame in the CNI (`cni_address`) and the size of the frame (`bytes` and others).

The association `Frame_in_R_Slot` defines the round-slots transmitting the frames.

`TTA.Message`

The class `TTA.Message` defines the attributes of a message transmitted. These attributes define, for example, the initialization value (`init_value`), the message type (`msg_type`),the validity span (`validity_span`), and whether a sender-status bit is transmitted with the message (`sender_status`). *Note:* Although it is possible to specify a replica-deterministic agreement algorithm as a message attribute (`agreement`) as well, it is strongly recommended to specify the agreement as an attribute of the associated `TTA.Msg_Type` instead.

The associations of the class `TTA.Message` define the temporal pattern of the transfer of messages, the subsystem sending the message, the data type of the message, and the frames carrying the message on the bus.

A few details about the initialization value

When using structured data types (such as arrays or structs), whether as global or local messages, the attribute `init_value` can be used in two ways:

1. The `init_value` is just one value. In that case, each element of the structured type will be initialized with that value.
   Example 1: a message of type struct `s_type` {int a; int b; int c} can have an `init_value` of 5. In this case, the initial values will be: a = 5; b = 5; c = 5.
2. The `init_value` is a Python tuple. In that case, the n-th element in the structured type will be initialized with the n-th item of the tuple.
   Example 2: the same message as above, `s_type` {int a; int b; int c} can have an `init_value` of (1,2,3). (*Note:* a Python tuple is normally indicated by round brackets and comma-separated items.) In this case, the initial values will be: a = 1; b = 2; c = 3.

Note that, in case of more complex structured data types, e.g., structs of structs, the tuple of initial values must still be a *flat* list of values, without any additional structuring brackets etc.

Example 3: assume that the message `s_type` is not a simple struct, but an array of three structured elements:

{struct a; struct b; struct c}

Assume further, that each struct element consists of two fields:

a {m;n}, b {o;p}, c {q;r}

To set the `init_value` of these fields to 0-5, respectively, the correct notation would be:

(0, 1, 2, 3, 4, 5)

resulting in m = 0; n = 1; o = 2; p = 3; q = 4; r = 5.

Special cases: If the tuple has fewer items than the structured type has elements, the remaining elements are initialized with zero. If the tuple has MORE items than the structured type has elements, the surplus tuple items are ignored and no error is displayed.

Important restriction: for single initialization values, the `init_value` can be a number or a name (like `SAFE_VALUE`). For `init_values` that are a tuple, only numbers are possible, but no names. (*Note:* It is NOT possible to use C-style initializers, e.g., {1,2,3} with curly brackets, as `init_value`!)

It should also be noted that the initialization code for global messages is generated with the function `_message_init()`, whereas the initialization code for local messages is generated with the function `_message_init_local()`.[56]

`TTA.Msg_Box`

`Msg_Boxes` can be used to group messages together for transmission over the bus. For example, you can create two `Msg_Boxes` `mb_x` and `mb_y`, and then put the messages i, j, and k into `mb_x`, and the messages u, v, and w into `mb_y`. To link a set of messages to a `TTA.Msg_Box`, use `TTA.Message_in_Msg_Box`. *Note:* `Msg_Boxes` can not only contain messages, but also other `Msg_Boxes`. A `TTA.Msg_Box` behaves like a single message in that it is linked to a subsystem and a cluster (`TTA.Subsystem_sends_Message` and `TTA.Cluster_uses_Message`), and that the scheduler links each `TTA.Msg_Box` to an N-frame (`TTA.Message_in_N_Frame`).

Within each `TTA.Msg_Box`, the scheduler places the messages in a specific order, similar to how it places messages (and `Msg_Boxes`) in N-frames. To do this, it does not need to know the contents of each `TTA.Msg_Box`, only its length.

A `TTA.Msg_Box` always occupies an even number of bytes in the frame (with `word` granularity[57]). This can lead to an inconsistent display, if the user specifies an odd length for

[56] This is because local messages do not need TTP, whereas global messages are TTP dependent. It is just a way to keep things clearly structured.

[57] This depends on the controller used, mostly the granularity is 2 bytes.

a TTA.Msg_Box (for example, 3 bytes): the user-specified length is displayed in the attribute editor of the tool, but in the schedule editor (see Section 3.10) the TTA.Msg_Box is displayed as it appears on the bus (i.e., with a length of 4 bytes). The 'Selected Messages' area, however, always shows its true length.

Msg_Boxes can be used as follows:

1. **Grouping of messages**
   The grouping of messages in Msg_Boxes (see example above) ensures that a set of messages is always transmitted together in the same frames, which may sometimes be desirable.
   Under certain conditions – specifically, if there are no other messages with a redundancy degree of 1–, if you define mb_x and mb_y with a redundancy degree of 1 and with the same size, phase, and period, they will end up on different channels.
2. **Optimization and tool performance**
   If several messages can be grouped together in Msg_Boxes, the number of links in the object model is reduced considerably, especially when there are lots of small messages. This enhances tool performance, i.e., reduces processing time and memory consumption. In addition, TTP-Build performance can be improved if the Msg_Boxes used are *homogeneous*, because only then the reduced number of access steps (FT-tasks) also reduces the amount of FT-COM code required. Homogeneous implies that all messages of a TTA.Msg_Box are:
   a) "FT-boxable", i.e.,
      - have an agreement of RD_1_valid (see Section 7)
      - have a length less than or equal to 8 bits
      - do not have a sender_status (see description in Section 12.2)
   b) non-multiplexed, i.e.,
      - they are transmitted in every generation of the TTA.Msg_Box
      - have the same redundancy degree as the TTA.Msg_Box itself

      In current versions of TTP-Plan the two corresponding attributes p_factor and redundancy_degree are view-only and have suitable defaults to force messages into being non-multiplexed. Only if messages are non-multiplexed can the corresponding TTA.Message_in_N_Frame links be omitted.
3. **Multilevel design** (not yet supported by current versions of TTP-Tools)
   Msg_Boxes extend the two-level design approach (cluster design and node design, see Section 1) with an intermediate level of *host group design*. This means you can design your cluster schedule with Msg_Boxes, thus reserving bandwidth without knowing anything about the messages actually transmitted. Then somebody else (e.g., a supplier) may go ahead and place messages into the Msg_Boxes by grouping them together. Since the messages are not directly "visible" on the bus, this is an important aspect considering intellectual property rights (IPR). The node designer will then assign additional information to the different nodes/hosts.

TTA.Msg_Type

The class TTA.Msg_Type defines the attributes of the data type of a set of messages. The available data types can be classified as follows:

1. **Atomic** (or "primitive") data types (Msg_Type_P): These are read by the CPU in one single access. The following types – corresponding to data types available in the C programming language – are available (in different lengths):
   - Integer (INT)
   - Unsigned Integer (UINT)
   - Float (FLOAT)

   The user-defined attributes of a Msg_Type_P are:
   - length: the length used by a message in a frame. This length can be shorter than the length the message occupies in the memory of a host if the message is transmitted over the bus in compressed form.
   - type_cat: the category of type

   Other attributes define the length used by the message in RAM (type_length)[58], the C typedef for the message, the replica-deterministic agreement (agreement), and whether the type is structured or atomic (is_atomic).

   If you define an atomic data type where the typedef is identical to the message type, no typedef will be generated in the message header file! To use such a data type, you have to make sure that the specified message type exists. You can achieve this in either of the following ways:
   - use a message type that is provided by the TTTech BSP (Board Support Package)
   - use a message type that is supported by your compiler
   - provide the typedef in your application code (because it has to be available to the code generated by TTP-Build later on)

   Atomic data types, unlike the *structured* data types described below, do not contain further data types. Therefore a Msg_Type_P does not require any link to describe the data type. Its only association is TTA.Message_uses_Msg_Type, defining the messages using this message type. The links of this association can be created by selecting the desired type in the TTA.Message attribute msg_type.
2. **Structured** (or complex) data types. The following two types are available:
   a) **Arrays** (Msg_Type_A):

      Arrays are used for a fixed number of elements of the same type. An array may, for example, consist of several integers (corresponding to a mathematical vector). There are onedimensional arrays, such as lists, but even multidimensional ones. Usually, arrays consist of large numbers (up to thousands) of elements. This is very useful, e.g., if you need to define a set of lines, you do

[58] This attribute corresponds to the result returned by the C function sizeof.

not need to define each line by defining each of its values, but you define one array for each line (since one line is just a set of data of the same type).

A `Msg_Type_A` object requires only one `TTA.Msg_Type_A_uses_Msg_Type` link. This is because, although one element may contain further elements, all elements in an array are, by definition, of a single data type.

b) **Structs** (`Msg_Type_S`):

Structs are logical units of a fixed number of elements of potentially different types; for example, a set of data describing a certain state (time, temperature, speed, on/off, etc.). Normally, the number of elements is limited to a few, and each element of the structure is referred to by a unique name. Structs simplify application programming because you only need to define and process a struct, not each single element.

In contrast to `Msg_Type_A`, a `Msg_Type_S` object requires several links of `TTA.Msg_Type_S_uses_Msg_Type` – one per structure element.

All objects of the class `TTA.Msg_Type` require links of the association `TTA.Message_uses_Msg_Type` defining the messages and complex data types using the message type.

Although the message type ultimately maps to a data type of the C programming language, it is good programming practice to provide more semantic information about the properties of the associated messages.

Because in C a data type is little more than the specification of a data representation (e.g., integer with 16 bits length), objects with very different semantics commonly share the same type. For a cluster design, it is strongly recommended to define message types according to the message semantics. Messages with different semantics should be associated to different message types – even if these message types map to the same representation in C. Defining message types according to message semantics offers several advantages, specifically:

- To conserve communication bandwidth, one might like to transmit as few bits as possible per message. How many bits are needed is dependent on message semantics. If messages use semantics-specific types, it is easy to turn compression on and off.
- The message type can provide visualization parameters for TTP-View:

  For each message type, a set of attributes specifies the visualization characteristics of messages using the type for TTP-View. These attributes are optional, but if TTP-View is to be used for monitoring the cluster, it is strongly recommended to specify values for them.

  You can, of course, set or change the values of all visualization parameters in TTP-View – but there you have to specify them per *message*, not per *type*; this means, if you want to visualize four different messages of the same type, you have four times as much work to do in TTP-View as in TTP-Plan or TTP-Build.
- The definition of replica-deterministic agreement algorithms for structured messages is made substantially easier by using semantics-specific types.

Note: if you do not specify the agreement either here or in `TTA.Message`, the default `RD_1_valid` will be used.

`TTA.N_Frame` (view-only)

The class `TTA.N_Frame` defines the attributes of an N-frame transmitted on the bus. These attributes include the address of the frame in the CNI and the size of the frame. The associations of the class `TTA.N_Frame` define the round-slot transmitting the frame and the messages carried by the frame. They are generated during scheduling and are view-only.

`RPV.Group`

In the attributes of this class the parameters for Remote Pin Voting (see Section 9) can be set for the group of RPV nodes known as the `RPV.Group`. Except for its specific attributes, an RPV group behaves like a normal subsystem (indicated by the view-only association `RPV.Group_is_a_Subsystem`).

`TTA.R_Slot` (view-only)

The class `TTA.R_Slot` defines the attributes of a round-slot of the cluster schedule. A round-slot occupies a specific position in the cluster schedule which is characterized by the round number and the properties of the associated slot.

`TTA.Slot`

The class `TTA.Slot` defines the attributes of a slot of the cluster schedule. Each slot occupies a specific position in the cluster schedule. This position is characterized by the phase relative to the round, and the duration of the slot.[59]

The attributes of a slot fall into two categories:

- *Communication:* these attributes define the parameters of the communication between the hosts of the cluster. They include the round-phase, the duration, the sequence number of the slot and of the position of the slot in the round.
- *Protocol:* these attributes configure the workings of the protocol in the cluster. They include the name of the sender of the slot and parameters for the distributed clock synchronization performed by TTP.

Although all input parameters for the `Slot` objects are optional, it is worth noting that the sequence of the slots in the round can be influenced with the `sort_key` attribute.

The associations of the class `TTA.Slot` define the host using the slot and the round-slots assigned to the slot. The number of round-slots used by a slot is determined by the number of rounds.

[59] The round-phase of a slot is equal to the sum of the phase and duration of the preceding slot.

TTA.Subsystem

At the cluster level, the class TTA.Subsystem is a black box with only a few optional attributes – its only role is to serve as a unit of distribution/replication and for packaging messages, i.e., it is a token used in associations. The associations of the class TTA.Subsystem define the hosts executing the subsystem and the messages sent by the subsystem. At the node level, designed in TTP-Build), the subsystems acquire a number of attributes which are irrelevant at the cluster level.

TTA.Web

A Web is a set of hosts connected by an independent physical medium, for example a bus. All the webs present in a system are assumed to be synchronized to each other (see Web synchronization). A web object named <cluster-name>_web with two channels is created automatically when a new Cluster object is defined. If you define additional webs, you may need to add TTA.Host_in_Web links as well (see Host_in_Web in Section 11.2). For the first web, this link is created automatically, as are TTA.Cluster_uses_Web and TTA.Web_uses_Channel.

A web's attributes are mainly designed for the specification of different aspects of data transmission, i.e., the byte order used for transmission (byte_order) and the transmission speed (transmission_speed). Also the physical interface is specified here (physical_interface). The tr_period must be identical in all webs of a cluster (for synchronization), but the transmission speed may differ.

With the attribute use_single_cluster_mode, the web can be restricted to one single mode of communication. By default, TTP-Plan supports two predefined cluster modes – a *startup* and a *working* mode. In single-mode operation, there is only one working mode, no mode changes are possible.

In a cluster with several webs (*multiweb cluster*), a node needs as many communication controllers as the number of webs it participates in (because you cannot connect one controller to different buses, for example), and it can only send in these webs. To extend the communication, a *gateway* function can be used that transfers data between webs. Such gateways can only be defined after the whole communication has been designed.

Note: downloading and monitoring of multiweb clusters is only possible on a web-by-web basis, i.e., for one web at a time. Therefore you need to specify the desired (*active*) web in the cluster attribute active_web before scheduling. This enables TTP-Plan to generate the corresponding TTA.Host_in_Web link and Monitor MEDL for this web.

If you intend to download and/or monitor all webs of the cluster, it is recommended to select each web in turn, make a schedule and save the resulting CDB and DDB with unequivocal names before proceeding to the next web. This way you get a collection of DDBs and matching CDBs for all webs, whereas otherwise only the last ones of the generated databases will "survive" (because the default DDB directory is overwritten with each scheduling run).

Most clusters will use only one web, but a multiweb cluster provides additional possibilities regarding safety and cost efficiency. For example, if some nodes are being shared by

different webs, the risk of system failure is reduced in a similar way as with multiple channels between one set of nodes. You may also use different cable types in different webs to save cost; for example by using cheaper ones where the transmission speed does not play a major role, while using high-speed cable for fast and safety-critical connections. This helps you avoid so called “common-mode errors”.

One slot does not necessarily need to be mapped to one host; it can be used by different hosts in different webs. Also a node that participates in more than one web can send different contents to its webs in one slot. Therefore the “sender” information of a message is not sufficient – if the transmission of the message is not fully redundant (i.e., transmitted on *all* channels), a receiving node that uses the “unused” channels can be “forgotten”and does not receive the message.[60] To avoid this, the sender of a message must be linked to the used channel *and* web, which is done in the `TTA.Host_in_Web` attribute `uses_channels`. In `TTA.Cluster_uses_Message` you can specify on which channel(s) the message may be transmitted, independent of the sending host.

Web synchronization

A real time-triggered architecture (TTA) only makes sense if the webs of a cluster are synchronized.[61] Two types of synchronization can be distinguished here:

- **Round synchronization:**
  The round *starts* of the webs as such are synchronized (with precision PI), but that does not necessarily mean the start of the *same* round in all webs is synchronized.
- **Cluster cycle synchronization:**
  The cluster cycles of the webs are synchronized such that the starts of all “first rounds” lie within precision PI. Normally this implies even synchronization at the round level, i.e., that all subsequent round starts are synchronized as well (round synchronization).

`TTA.X_Frame` (view-only)

The class `TTA.X_Frame` defines the attributes of an X-frame transmitted on the bus. These attributes include the address of the frame in the CNI and the size of the frame.

The associations of the class `TTA.X_Frame` define the round-slot transmitting the frame and the messages carried by the frame.

11.2. Associations

This section presents all associations of TTP-Plan’s object model. A more detailed description of the attributes of these associations is given in Sections 12 to 12.3.

[60] The problem with the redundancy level is that “2” can mean A and B in the same web, or A+A in both webs etc.

[61] This is done externally by the OS (“external rate correction”) and may take quite a long time.

TTA.Cluster_uses_Message

The association TTA.Cluster_uses_Message defines the temporal pattern for the exchange of messages in the cluster. The attributes of TTA.Cluster_uses_Message specify the period and redundancy with which a message is sent on the bus, and constraints on the rounds in which the message is sent.

TTA.Cluster_uses_Web (view-only)

Each web has to be linked to the TTA.Cluster object. The links of this association are generated automatically when a cluster is defined or when new webs are created.

Frame_in_R_Slot (view-only)

The association Frame_in_R_Slot defines the round-slot in which a frame is transmitted.

TTA.Host_in_Cluster

The association TTA.Host_in_Cluster defines the hosts comprising a cluster. The attribute is_foreign specifies whether the node is synchronized with the cluster.[62] TTP-Plan creates the links of this association automatically during the creation of a host, if the cluster object is already defined.

TTA.Host_in_Web

If there is only one web in the cluster, all hosts are automatically connected to it by means of TTA.Host_in_Web links. For additional webs and hosts this automatism depends on the order of definitions:

- If you create additional webs while there are no hosts present, and *then* define hosts, all hosts are automatically linked to all webs.
- If you define some hosts and *then* define additional webs, you need to create the Host_in_Web links for the new webs manually.

An optional, but important, attribute of Host_in_Web is uses_channels, where you can specify which channels each host should be connected to. This is decisive for the transmission of messages to different webs.

[62] If set to 'yes', the node acts as a gateway.

TTA.Host_runs_Subsystem_in_Cluster

The association TTA.Host_runs_Subsystem_in_Cluster defines the distribution of software components in the cluster. In a cluster, each host executes a specific set of subsystems – this set is specified by the links of TTA.Host_runs_Subsystem_in_Cluster.

Note: a subsystem is replicated in a certain cluster if it has links to more than one host.

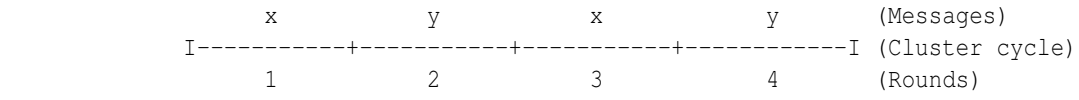
TTA.Host_uses_Slot

The association TTA.Host_uses_Slot defines the slots used by the hosts of a cluster. This is a one-to-one association without editable attributes.

TTA.Message_depends_on_Message

With the association TTA.Message_depends_on_Message you can define dependencies between messages, i.e., if message x is needed to generate message y. Furthermore, you can make restrictions concerning the *generation lag*.

All global messages have to have exactly one sender task each, whereas they may have several receiver tasks. This is defined in the association TTA.Task_uses_Message (see the TTP-Build manual for details). However, this association cannot model whether any real "flow of information" takes place. Consider the following example:



If the input of a cluster cycle is a message x, and the output is a message y, their sequence may suggest that that y depends on x, but they may as well be completely independent, no information about this is given in TTA.Task_uses_Message. If there *is* a dependency, this has to be defined at the cluster level in TTA.Message_depends_on_Message (for all nodes!), but no details are given there ("global control circuit"). To be declared dependent, x and y must have the same period and their validity spans must overlap, else an invariant will be raised during scheduling.

TTA.Message_depends_on_Message tells the scheduler the order of the application tasks and FT-tasks that access these messages. Important in this context is the generation_lag parameter: it gives information about the temporal connection between the messages (there is no rule that says m1 must be processed at once), i.e., the allowed delay between input and output. *Note:* this delay must be the same on all nodes to ensure replica determinism! The generation lag is given in the number of message generations. *Note:* the maximum generation lag *must* be less than the number of message generations per cluster cycle!

- 0: as little delay as possible; x produces the earliest possible y
- 1: one message generation allowed between x and y
- 2: two message generations allowed between x and y; in the above example, where each message is only sent twice per cycle, this would mean y could also be produced in the next cluster cycle. This is not allowed (maximum generation lag equals number of message generations), so "2" is not a valid value.

TTA.Message_depends_on_Message influences both the cluster schedule and the node schedule. Its links put rather severe constraints on the scheduler, i.e., it gets more difficult to find a valid schedule because there has to be sufficient time/space for the tasks to be run between the linked messages. Hence the system designer is strongly advised to consider the dependencies in the system very carefully, so as not to render scheduling impossible or suboptimal.

Some general considerations:

- The FT-task receiving x has to run *after* receiving, whereas the FT-task sending y has to run *before* sending. If the specified generation_lag is small, this leaves only a very narrow processing window.
- If you already placed the dependent messages in the "correct" order according to the desired flow of information, you still need to define TTA.Message_depends_on_Message links for them. Otherwise the scheduler does not know about these dependencies and may still place them differently!
- You can also use TTA.Message_depends_on_Message to let a message depend on itself, e.g., x depends on x. In this case you have to set the generation_lag larger than "0". This is especially useful if you want to create a counter reaching across several tasks. For example:

  ```
  x ---> Task1 ---> local message ---> Task2 ---> x
  ```

  Here the order of tasks is very important and must not be reversed, hence it makes sense to define a dependency between their messages.

TTA.Message_in_Msg_Box

This association links a message to a message box.

TTA.Message_in_N_Frame

The association TTA.Message_in_N_Frame defines the N-frames carrying the messages exchanged between the hosts of a cluster. The attributes of this association specify the position of the message in the N-frame.

TTA.Message_uses_Msg_Type

The association TTA.Message_uses_Msg_Type defines the C data types of the messages exchanged between the hosts of a cluster. The links of this association can either be cre-

ated directly in the link selector, or by setting the `TTA.Message` attribute `msg_type` (so called *attribute-based links*).

`TTA.Msg_Type_A_uses_Msg_Type`

This association defines the structure of "array" message types. Its only required attribute is `bounds`, where you have to specify the number of elements in the array.

`TTA.Msg_Type_S_uses_Msg_Type`

This association defines the structure of "struct" message types. Its only required attribute is `comp_name`, where you have to specify the names of the structure elements.

`TTA.Slot_has_R_Slot` (view-only)

The links of this association are generated by the scheduler and define which round-slots belong to which slot.

`TTA.Subsystem_sends_Message`

The association `TTA.Subsystem_sends_Message` defines the subsystems sending the messages exchanged between the hosts of a cluster. The attributes of this association are defined in TTP-Build.

`TTA.Web_uses_Channel` (view-only)

The links of this association are generated automatically when a web is defined. By default there are two channels per web. For single-channel operation of a web, one channel has to be removed manually and the replication and redundancy settings must be adapted accordingly. *Note:* Independent of which channel is removed, the remaining *physical* channel is always 1!

12. Object Model Attributes

In TTP-Plan there are two categories of input data: *required* (application-specific) and *optional* (solution-specific) attributes.

Required input data are essential application-specific attributes which must be specified before a cluster schedule can be generated. A specific set of these application-specific attributes is compatible with many different sets of solution-specific attributes.

Optional input data are solution-specific attributes which characterize one specific design solution for a set of application-specific attributes. For these optional attributes, TTP-Plan uses default values, unless the cluster designer specifies them differently. By providing values for some (or all) solution-specific attributes, the cluster designer can control the properties of the generated cluster schedule in fine detail.

Based on the input data, TTP-Plan generates a cluster schedule. The TTP-Plan scheduler adds some internal attributes to the classes and associations described above and creates slots and frames.

Each host sends its messages during its own slot. A slot is characterized by its phase relative to the round and by its duration. For each slot, two frames are transmitted per round.[63]

Each frame is sent in a specific slot in a specific round on a specific channel of the bus. There are three kinds of frames: an I-frame carries information used by the protocol. An N-frame carries messages. An X-frame carries information used by the protocol as well as messages.

TTP-Plan provides a schedule editor which lets you move messages to different N-frames/X-frames and rearrange the phases of the slots (see Section 3.10).

12.1. Required Input Data

`RPV.Message`: required attributes

Scope: TTP˙Plan scope

[63] Strictly speaking, one frame per round and channel.

➤ Name `init_value`
Type `Text`
Description Initialization value of the message (in ASCII characters).
For structured data types (such as arrays or structs or combinations thereof) the `init_value` can either be a single value or a Python tuple (numbers only!), e.g., (1,2,3), without additional brackets etc. In case of a single value, all elements of the message's data type are initialized with this value. In case of a tuple, the n-th element of the data type is initialized with the n-th item of the tuple. Please refer to the user manual for a more detailed description.
This value is important for receivers of the message running in the first cluster cycle before it is transmitted on the bus for the first time.
Normally, a local receiving task runs only after the corresponding sending task has been executed. However, especially in the case of cyclic dependencies it is important that one task is allowed to start without receiving its message first. The setting of the `init_value` is strongly application dependent and must be carefully considered by the designer.

➤ Name `msg_type`
Type `Link`
Description `Msg_type` linked to `Message`.

`TTA.Checksum`: required attributes

Scope: TTP Plan scope

➤ Name `length`
Type `Bit_Size`
Syntax Specified as `[bytes][:bits]`.
Examples:

- 2 = two bytes
- 1.5 = 12 bits
- :1 = one bit
- 1:2 = 10 bits

If you set this attribute via a script, and pass an integer to the `set` function (e.g., `raw=0`), you must specify a length in bits.
Note: regular expressions are allowed in raw values (i.e., in the interactive editor and in scripts calling `set_raw`) and will be interpreted as a (possibly fractional) number of bytes.
Description Length of `Checksum` (in bytes:bits).

`TTA.Cluster`: required attributes

Scope: TTP˙Plan scope

➤ Name `tr_period`
Type `Micro_Time_Span`
Syntax The default unit is us. If you specify another unit, it must be separated from the number by at least one space. You can use the following units: ms, ns, ps, s, us.
Description Requested duration of round (in microseconds).
This field specifies the duration of the smallest recurring communication pattern (round). This communication pattern consists of a sequence of slots. In each slot only a single host is allowed to send (on one channel).

`TTA.Cluster_uses_Message`: required attributes

Scope: TTP˙Plan scope

➤ Name `d_period`
Type `Micro_Time_Span`
Syntax The default unit is us. If you specify another unit, it must be separated from the number by at least one space. You can use the following units: ms, ns, ps, s, us.
Description Design period, i.e., the maximum admissible transmission period of the message (in microseconds).
The scheduler then chooses an actual transmission period `a_period` which is shorter than or equal to `d_period` and is equal to `TTA.Cluster.tr_period` times a power of two (1, 2, 4, ...).
Exception: if `fixed_round_number` is specified for the cluster, the `d_period` must be a divisor of `TTA.Cluster.fixed_round_number * TTA.Cluster.tr_period`.

`TTA.Host_in_Cluster`: required attributes

Scope: TTP˙Plan scope

➤ Name `serial_number`
Type `Int`
Description Number to address this node for download (must be unique in the cluster).
This field specifies a number that has to be unique within the cluster. It is solely used during download to address this node unambiguously.

TTA.Message: required attributes

Scope: TTP Plan scope

➤ Name `init_value`
Type `Text`
Description Initialization value of the message (in ASCII characters).
For structured data types (such as arrays or structs or combinations thereof) the `init_value` can either be a single value or a Python tuple (numbers only!), e.g., (1,2,3), without additional brackets etc. In case of a single value, all elements of the message's data type are initialized with this value. In case of a tuple, the n-th element of the data type is initialized with the n-th item of the tuple. Please refer to the user manual for a more detailed description.
This value is important for receivers of the message running in the first cluster cycle before it is transmitted on the bus for the first time.
Normally, a local receiving task runs only after the corresponding sending task has been executed. However, especially in the case of cyclic dependencies it is important that one task is allowed to start without receiving its message first. The setting of the `init_value` is strongly application dependent and must be carefully considered by the designer.

➤ Name `msg_type`
Type `Link`
Description `Msg_type` linked to `Message`.

TTA.Message_depends_on_Message: required attributes

Scope: TTP Plan scope

➤ Name `generation_lag`
Type `Int`
Description Delay between the generation of the predecessor message and the generation of the successor message.
If a message X depends on another message Y (which means that the value of X is calculated based on the the value of Y), the generation lag defines the delay of this calculation. A generation lag of 0 means that the value of generation n of message X is calculated based upon the value of generation n of message Y. A generation lag of 1 means that the value of generation n of message X is calculated based upon the value of generation n-1 of message Y. Please refer to the manual for a more detailed description of this. Be aware that the generation lag must be smaller than the number of generations that are transmitted within one application cycle.

TTA.Msg_Box: required attributes

Scope: TTP Plan scope

➤ Name `data_size`
Type `Bit_Size`
Syntax Specified as `[bytes][:bits]`.
Examples:

- 2 = two bytes
- 1.5 = 12 bits
- :1 = one bit
- 1:2 = 10 bits

If you set this attribute via a script, and pass an integer to the `set` function (e.g., `raw=0`), you must specify a length in bits.
Note: regular expressions are allowed in raw values (i.e., in the interactive editor and in scripts calling `set_raw`) and will be interpreted as a (possibly fractional) number of bytes.
Description Length of message data carried by `Msg_Box` (in bytes).
Must be large enough to carry all messages in this `Msg_Box`. Spare length can be used for later schedule steps.

TTA.Msg_Type_A_uses_Msg_Type: required attributes

Scope: TTP Plan scope

➤ Name `bounds`
Type `Int_List`
Description Number of elements of the array (can be a list of bounds for each dimension, or a single value).
Since an array can be multidimensional, you have to specify the 'length per dimension', e.g., in the form x,y or (x,y), with the values in the same order as the links are to be created. This is important for the later access to the data in C code because the order of elements in the array corresponds to the indices in the code.

TTA.Msg_Type_P: required attributes

Scope: TTP Plan scope

➤ Name `length`
Type `Bit_Size`
Syntax Specified as `[bytes][:bits]`.
Examples:

- 2 = two bytes
- 1.5 = 12 bits
- :1 = one bit
- 1:2 = 10 bits

If you set this attribute via a script, and pass an integer to the `set` function (e.g., `raw=0`), you must specify a length in bits.
Note: regular expressions are allowed in raw values (i.e., in the interactive editor and in scripts calling `set_raw`) and will be interpreted as a (possibly fractional) number of bytes.
Description Significant length of a single message of this type, i.e., the space that one message of this type occupies in a frame (in bytes:bits). On a CPU, messages usually have the length of a word (e.g. 2-byte word, 4-byte word) for fast access, whereas on a bus, messages should be as short as possible to optimize bandwidth. A 12-bit sensor value should have a `length` of 12 bits, even though it might be stored in a 16-bit variable. The communication layer provides the conversion between those representations.

➤ Name `type_cat`
Type `Type-Cat`
Description Category of type.
The `type_cat` specified must match the `type_cat_implied` as implied by the `typedef` attribute (if defined) of the `Msg_Type`.
INT = signed integer value,
UINT = unsigned integer value,
REAL = floating point value.
This attribute defines the major type categories that are then subdivided into more specific categories. For example, the category INT contains types like `word`, `byte`, etc. Note: the tool might accept a user-defined `typedef`, but to avoid compiler errors, the type-related attributes have to match in any case!

`TTA.Msg_Type_S_uses_Msg_Type`: required attributes

Scope: TTP Plan scope

➤ Name `comp_name`
Type `Name`
Description Name of struct element.
Each struct element needs a unique name which is then used to access it. This name is specified in the attribute `comp_name`.

TTA.Sched_Step_CS: required attributes

Scope: TTP Plan scope

➤ Name `msg_type`
Type `Link`
Description `Msg_type` linked to `Message`.

TTA.Subsystem_Status_Message: required attributes

Scope: TTP Plan scope

➤ Name `init_value`
Type `Text`
Description Initialization value of the message (in ASCII characters).
For structured data types (such as arrays or structs or combinations thereof) the `init_value` can either be a single value or a Python tuple (numbers only!), e.g., (1,2,3), without additional brackets etc. In case of a single value, all elements of the message's data type are initialized with this value. In case of a tuple, the n-th element of the data type is initialized with the n-th item of the tuple. Please refer to the user manual for a more detailed description.
This value is important for receivers of the message running in the first cluster cycle before it is transmitted on the bus for the first time.
Normally, a local receiving task runs only after the corresponding sending task has been executed. However, especially in the case of cyclic dependencies it is important that one task is allowed to start without receiving its message first. The setting of the `init_value` is strongly application dependent and must be carefully considered by the designer.

➤ Name `msg_type`
Type `Link`
Description `Msg_type` linked to `Message`.

TTA.Web: required attributes

Scope: TTP Plan scope

➤ Name `byte_order`
Type `Byte_Order`
Description Byte order used for the transmission of data on the bus.
This attribute specifies the byte order that is used on the bus. Each host may have a different byte order, but if the byte order of the host is different from the byte order on the bus, or if the host interface causes byte swapping, a conversion has to be done by the communication layer. This conversion may be time-consuming.

➤ Name `transmission_speed`
Type `Int`
Description Speed of data transmission in kbit/s.
This field specifies the speed of data transmission. It depends on the clock speed of the communication controller and a prescale-factor. It further depends on the physical layer used. This number constitutes the nominal bandwidth for transmission.

`TTP.Clock_Sync_Multiweb`: required attributes

Scope: TTP Plan framework scope

➤ Name `max_web_deviation`
Type `Micro_Time_Span`
Syntax The default unit is us. If you specify another unit, it must be separated from the number by at least one space. You can use the following units: ms, ns, ps, s, us.
Description The maximum tolerable deviation between two webs (in microseconds).

`TTP.Host_on_MHub`: required attributes

Scope: TTP Plan scope

➤ Name `cable_length`
Type `Float`
Description Length of cable (in m).

12.2. Optional Input Data

`ESC.Physical_Interface_MFM`: optional attributes

Scope: TTP Plan framework scope

➤ Name `ignore_noise`
Type `Boolean`
Default `no`
Syntax The following string values are accepted as valid Boolean values: 0, 1, f, false, n, no, t, true, y, yes
Description Select parameters for ignoring noise. See Application Note 134 Rev.1.6.

`ESC.Physical_Interface_Manchester`: optional attributes

Scope: TTP Plan framework scope

➤ Name `rs485`
Type `Boolean`
Default `no`
Syntax The following string values are accepted as valid Boolean values: 0, 1, f, false, n, no, t, true, y, yes
Description If this value is set to true, the logical level 0 is accepted as idle. This should be set for unbiased RS485 physical layers.

➤ Name `stabilization_duration_ns`
Type `Int`
Default `500`
Description Duration until the physical layer is stable, i.e. until the preamble is undisturbed. See Application Note 134.

`RPV.Group`: optional attributes

Scope: TTP Plan scope

➤ Name `decision_point`
Type `RPV_Decision_Spec`
Default `automatically computed`
Syntax The syntax for specifying slots is:

```
(slot_i, slot_j, ...) every k from l.
```

There may be more than one slot specification in the decision point specification, separated by semicolons, e.g.,

```
(1,2) every 1 from 2; (3,4) every 4 from 3
```

The decision is taken at each round-slot in the set (the slot numbers are shown in the schedule editor, the first slot in the round has the number zero) starting with round `l` and repeated every `k` rounds. If only a single slot is specified, the parentheses can be omitted.
A slot number of -1 specifies the last slot in the round, -2 specifies the next-to-the-last slot...
The clauses `every k` and `from l` are optional. The default for `k` is 1, meaning `in every round`. The default for `l` is 1, meaning `starting with the first round`. Setting l to -1 specifies the last round of the cluster cycle.
For example, `-1 every 1 from -1` specifies the last round-slot of a round, which may be a reasonable default.
Description Specification of the round-slots in which RPV decisions take place (see syntax description).
The votes from different hosts get accumulated until this decision point is reached. At this point in time the votes are summed up and the desired decision is taken and the voting counter is reset.

- Name `init_value`
  Type `Boolean`
  Default `no`
  Syntax The following string values are accepted as valid Boolean values: 0, 1, f, false, n, no, t, true, y, yes
  Description Value used by controlling hosts immediately after startup.

- Name `permanence_delay`
  Type `Int`
  Default `0`
  Description Specifies the number of non-optional slots that must pass before a message of this subsystem is used by the receiver.
  To achieve full replica determinism, a message replica can be used by the receiver only when the sending node has been acknowledged by TTP. This can happen not earlier than 2 non-optional slots after the host's sending slot and not later than 1 round after the host's sending slot.
  0 - means no replica determinism in case of a transmission failure.
  2 - means replica determinism in case of exactly one transmission failure.
  2 + n - means replica determinism in case of a transmission failure and at most n additional failures of other hosts.
  Please note : The higher this value is set, the more failures are tolerated while still having replica determinism, but on the other hand, task scheduling becomes harder.

- Name `r_period`
  Type `Int`
  Default `1`
  Description Defines the period of the associated RPV message (in rounds). An `r_period` of 1 means the message is sent every round, whereas 2 means every second round, etc.

- Name `reintegration_timeout`
  Type `Int`
  Default `2`
  Description Only used if `reintegration_type` is `h_state_reintegration`. Specifies how many rounds this subsystems waits for a valid h-state before it starts up using the initial h-state.

- Name `reintegration_type`
  Type `Reintegration_Type`
  Default `Reinit_Reintegration`
  Description Specifies the type of reintegration required for this subsystem.

- Name `startup_quota`
  Type `Int`
  Default `None`
  Description Specifies the number of subsystems which must be active before the mode change from the TTP startup mode to the TTP application mode can be requested.

- Name `status`
  Type `Boolean`
  Default `no`
  Syntax The following string values are accepted as valid Boolean values: 0, 1, f, false, n, no, t, true, y, yes
  Description Specifies if this subsystem needs a dedicated subsystem status message to indicate if the subsystem is running correctly.

- Name `status_r_period`
  Type `Int`
  Default `automatically computed`
  Description Specifies the period of the associated status message (in rounds). An `r_period` of 1 means the message is sent every round whereas 2 means every second round, etc. (default: once per cluster cycle).

- Name `threshold`
  Type `Int`
  Default `automatically computed`
  Description The RPV logic trips if the number of voting messages (+"threshold˙bias") sent with value `1` received prior to the decision point is greater than the threshold.

- Name `threshold_bias`
  Type `Int`
  Default `0`
  Description After the RPV logic tripped (i.e., the number of RPV messages sent with value `1` exceeds the threshold) the threshold bias is added to the number of RPV messages (sent with the value `1`) received for the evaluation of the decision at the next decision point. This algorithm provides a hysteresis in the voting behavior.

- Name `use_full_h_state_as_guard`
  Type `Boolean`
  Default `no`
  Syntax The following string values are accepted as valid Boolean values: 0, 1, f, false, n, no, t, true, y, yes
  Description Specifies whether all h-state messages sent by this subsystem should be used as guard for the execution of the tasks belonging to this subsystem.

➤ Name `voting_logic`
Type `Boolean`
Default `yes`
Syntax The following string values are accepted as valid Boolean values: 0, 1, f, false, n, no, t, true, y, yes
Description If the voting logic is set to `yes` the RPV pin defaults to high and is set to low when the RPV logic trips (i.e., when the number of RPV messages sent with value `1` exceeds the threshold). Bear in mind that the number of received logical ones must exceed the `threshold` (i.e., be greater). The opposite applies for setting the voting logic to `no`. In this case the RPV pin defaults to low and will be driven high when the RPV logic trips.

`RPV.Message`: optional attributes

Scope: TTP Plan scope

➤ Name `agreement`
Type `RD_Agreement`
Default `automatically computed`
Description Replica-deterministic agreement algorithm.
This parameter is only necessary for non-replica-deterministic messages sent by replicated subsystems.

➤ Name `caption`
Type `Text`
Default `automatically computed`
Description Caption displayed by TTP-View instruments for objects of this type.

➤ Name `is_h_state`
Type `Boolean`
Default `no`
Syntax The following string values are accepted as valid Boolean values: 0, 1, f, false, n, no, t, true, y, yes
Description Specifies if the message carries the h-state of a replicated task.

➤ Name `max_prd`
Type `Micro_Time_Span`
Default `None`
Syntax The default unit is us. If you specify another unit, it must be separated from the number by at least one space. You can use the following units: ms, ns, ps, s, us.
Description Maximum post-receive duration (in microseconds): specifies the maximum duration between the end of transmission and the consumption of the message.

- Name `max_psd`
  Type `Micro_Time_Span`
  Default `None`
  Syntax The default unit is us. If you specify another unit, it must be separated from the number by at least one space. You can use the following units: ms, ns, ps, s, us.
  Description Maximum pre-send duration (in microseconds): specifies the maximum duration between the creation of the message and the start of transmission.
- Name `sender_status`
  Type `Boolean`
  Default `no`
  Syntax The following string values are accepted as valid Boolean values: 0, 1, f, false, n, no, t, true, y, yes
  Description Specifies if a sender-status bit is transmitted along with the message. A sender-status bit allows the sending host to tell the receivers of the message whether the message value is valid or not.
- Name `subsystem`
  Type `Link`
  Default `None`
  Description Subsystem that sends the message
- Name `validity_span`
  Type `Micro_Time_Span`
  Default `None`
  Syntax The default unit is us. If you specify another unit, it must be separated from the number by at least one space. You can use the following units: ms, ns, ps, s, us.
  Description Amount of time the message remains valid (microseconds).

`TTA.Channel`: optional attributes

Scope: TTP Plan scope

- Name `is_wired`
  Type `Boolean`
  Default `automatically computed`
  Syntax The following string values are accepted as valid Boolean values: 0, 1, f, false, n, no, t, true, y, yes
  Description Specifies whether this channel of the TTP-Controller is wired.
- Name `physical_interface`
  Type `Physical_Interface`
  Default `automatically computed`
  Description Physical interface to be used for this channel (overwrites the default `physical_interface` of the web).

`TTA.Cluster`: optional attributes

Scope: TTP Plan scope

➤ Name `active_web`
Type `Web`
Default `automatically computed`
Description Specifies the web used for downloading/viewing.

➤ Name `allocate_sender_status_at_end`
Type `Boolean`
Default `no`
Syntax The following string values are accepted as valid Boolean values: 0, 1, f, false, n, no, t, true, y, yes
Description Allocate sender status at end of `Frame` or `Msg_Box` (growing inwards), if true.

➤ Name `allocate_sender_status_at_start`
Type `Bit_Size`
Default `0`
Syntax Specified as `[bytes][:bits]`.
Examples:

- 2 = two bytes
- 1.5 = 12 bits
- :1 = one bit
- 1:2 = 10 bits

If you set this attribute via a script, and pass an integer to the `set` function (e.g., `raw=0`), you must specify a length in bits.
Note: regular expressions are allowed in raw values (i.e., in the interactive editor and in scripts calling `set_raw`) and will be interpreted as a (possibly fractional) number of bytes.
Description Allocate sender status at start of `Frame` or `Msg_Box`, in a segment of the size (in bytes:bits) as specified for this attribute.

➤ Name `alternate_channels`
Type `Boolean`
Default `yes`
Syntax The following string values are accepted as valid Boolean values: 0, 1, f, false, n, no, t, true, y, yes
Description If true, the scheduler will alternate the sequence in which channels are considered for message allocation. The alternation concerns different rounds of the same host, and the starting sequence for different hosts.

➤ Name `application_id`
Type `Int`
Default `0`
Description Identifier for this application.

➤ Name `application_version`
Type `Int`
Default `0`
Description Version of this application.

➤ Name `big_bang_enabled`
Type `Boolean`
Default `True`
Syntax The following string values are accepted as valid Boolean values: 0, 1, f, false, n, no, t, true, y, yes
Description In order to optimize the performance (i.e., the startup time) in case of cluster designs with a single coldstarter (typical Master-Hub (M-Hub) setup), it shall be possible to disable the big bang algorithm. This algorithm only makes sense if there is more than one coldstarter.
Plugin This attribute is only available for certain plugins

➤ Name `clock_sync`
Type `Clock_Sync`
Default `Clock_Sync_Standard`
Description Clock synchronization properties (to be edited in the corresponding object editor).

➤ Name `controller_type`
Type `Controller`
Default `TTTech_C2NF`
Description Type of communication controller used in this cluster.

➤ Name `max_frame_size`
Type `Bit_Size`
Default `<=>.protocol.max_frame_bytes`
Syntax Specified as `[bytes][:bits]`.
Examples:

- 2 = two bytes
- 1.5 = 12 bits
- :1 = one bit
- 1:2 = 10 bits

If you set this attribute via a script, and pass an integer to the `set` function (e.g., `raw=0`), you must specify a length in bits.
Note: regular expressions are allowed in raw values (i.e., in the interactive editor and in scripts calling `set_raw`) and will be interpreted as a (possibly fractional) number of bytes.
Description Maximum size of N˙Frame, specified as `[bytes][:bits]`.
For instance, `2` means two bytes, `:1` means one bit, `1:2` means 10 bits.

- Name `physical_interface`
  Type `Physical_Interface`
  Default `Manchester`
  Description Physical interface to be used.

- Name `allow_schedule_extension`
  Type `Boolean`
  Default `no`
  Syntax The following string values are accepted as valid Boolean values: 0, 1, f, false, n, no, t, true, y, yes
  Description If true, the scheduler will allow the extension of the schedule in discrete steps.

- Name `schedule_step_cs_bytes`
  Type `Int`
  Default `2`
  Description Number of bytes used for transmission of schedule-step checksums.

- Name `schedule_step_cs_period`
  Type `Micro_Time_Span`
  Default `automatically computed`
  Syntax The default unit is us. If you specify another unit, it must be separated from the number by at least one space. You can use the following units: ms, ns, ps, s, us.
  Description Period of messages carrying schedule-step checksums (in microseconds). Default: once per cluster cycle.

- Name `fill_rounds_round_robin`
  Type `Boolean`
  Default `automatically computed`
  Syntax The following string values are accepted as valid Boolean values: 0, 1, f, false, n, no, t, true, y, yes
  Description If set to `true`, the cluster scheduler will fill the rounds in a round-robin fashion. This results in similar frame lengths for all rounds of a host. If set to `false`, the cluster scheduler will fill the rounds top-down, i.e., it will only start filling another round when there is no place left in the current round.
  The default depends on whether any host has a `c_state_round_spec` or `use_x_frames`: if you specify any `c_state_round_spec` or `use_x_frames`, the default for `fill_rounds_round_robin` is `true`, otherwise it is `false`.

- Name `fixed_round_number`
  Type `Int`
  Default `None`
  Description Fixed number of rounds to use for cluster schedule.
  Normally, the cluster scheduler chooses how many rounds to use (the number chosen is always a power of two). If `fixed_round_number` is specified, the cluster scheduler will use exactly that number of rounds. All message periods must then be divisors of `fixed_round_number` and integer multiples of `tr_period`.
  Specifying a `fixed_round_number` which is not a power of two will result in schedules with less effective bandwidth than power-of-two schedules.

- Name `max_tdma_rounds`
  Type `Int`
  Default `16`
  Description Maximum number of rounds allowed.

- Name `min_tdma_rounds`
  Type `Int`
  Default `automatically computed`
  Description Minimum number of rounds allowed. The default is 2, if no fixed-round-number scheduling is used.

`TTA.Cluster_uses_Message`: optional attributes

Scope: TTP Plan scope

- Name `allow_channels`
  Type `Channel_List`
  Default `automatically computed`
  Description Specifies which channels can be used for the transmission of `message`.

- Name `max_round`
  Type `Int`
  Default `automatically computed`
  Description Latest round for the first message transmission in the cluster cycle. Use this and `min_round` to restrict the transmission of a message to specific rounds in the cycle.

- Name `min_round`
  Type `Int`
  Default `1`
  Description Earliest round in the cluster cycle where the message may be transmitted. Use this and `max_round` to restrict the transmission of a message to specific rounds in the cycle.

➤ Name `redundancy_degree`
Type `Int`
Default `1`
Description Number of channels to be used for transmission of this message.

➤ Name `sampling_factor`
Type `Int`
Default `1`
Description Number of transmissions per calculation of this message. This information is relevant for the task scheduling of the sender task in the node design. If a message is sent more often than the task is executed (i.e., the `sampling_factor` is greater than one), the same message copy will get transmitted several times (`a_period_max * sampling_factor == sender_task_period`).
1: send each computed value,
3: send every computed value 3 times.

➤ Name `sampling_phase`
Type `Int`
Default `0`
Description If `sampling_factor > 1`, the `sampling_phase` specifies the phase of the sending task. 0 means the task must run before the first transmission in the cluster cycle. If this value is 1, one copy of the message transmitted in the current cycle is still an old copy from the previous cycle, and so on.

`TTA.Controller_Set`: optional attributes

Scope: TTP˙Plan framework scope

➤ Name `uT_ns`
Type `Float`
Default `automatically computed`
Description Duration of a microtick (in nanoseconds).

`TTA.Host`: optional attributes

Scope: TTP˙Plan scope

➤ Name `SYF`
Type `Boolean`
Default `automatically computed`
Syntax The following string values are accepted as valid Boolean values: 0, 1, f, false, n, no, t, true, y, yes
Description If set to “yes”, this host is one of the actively synchronizing nodes. Note that, in this case, `allow_active_role` must also be set to “yes” and the `Host_in_Cluster` attribute `is_optional` must be set to “no”.

➤ Name `allow_active_role`
Type `Boolean`
Default `yes`
Syntax The following string values are accepted as valid Boolean values: 0, 1, f, false, n, no, t, true, y, yes
Description If true, this host is allowed to send. Otherwise it can only receive.

➤ Name `allow_mode_change`
Type `Boolean`
Default `yes`
Syntax The following string values are accepted as valid Boolean values: 0, 1, f, false, n, no, t, true, y, yes
Description If true, this host is allowed to issue a cluster-mode change request.

➤ Name `min_integration_count`
Type `Int`
Default `2`
Description To disable the communication blackout for one node, the `minimum_integration_count` (MIC) must be set to 1 (at least for some nodes). This is necessary because of the scenario where one node (X- or I-frame sender) is the only active node (and does not shut down because of the disabled communication blackout detection). In this case, only nodes with MIC=1 can integrate into the running cluster.
Plugin This attribute is only available for certain plugins

➤ Name `perform_clique_detection`
Type `Boolean`
Default `True`
Syntax The following string values are accepted as valid Boolean values: 0, 1, f, false, n, no, t, true, y, yes
Description In order to optimize operation of the M-Hub, this attribute allows to selectively disable clique and communication blackout detection for a node. This means it is possible to clear the PC (perform clique detection) flag in each round-slot entry. Usually it is set in the node's own sending slot.
Plugin This attribute is only available for certain plugins

➤ Name `rpv_startup_logic`
Type `Boolean`
Default `no`
Syntax The following string values are accepted as valid Boolean values: 0, 1, f, false, n, no, t, true, y, yes
Description Indicates whether the RPV pin should initially (i.e., until the first decision point) be set to high (`yes`) or low (`no`).

➤ Name `use_x_frames`
Type `Boolean`
Default `no`
Syntax The following string values are accepted as valid Boolean values:
0, 1, f, false, n, no, t, true, y, yes
Description Specifies whether the host should use X-frames instead of I-frames to transmit C-state information.

➤ Name `max_frame_size`
Type `Bit_Size`
Default `automatically computed`
Syntax Specified as `[bytes][:bits]`.
Examples:

- 2 = two bytes
- 1.5 = 12 bits
- :1 = one bit
- 1:2 = 10 bits

If you set this attribute via a script, and pass an integer to the `set` function (e.g., `raw=0`), you must specify a length in bits.
Note: regular expressions are allowed in raw values (i.e., in the interactive editor and in scripts calling `set_raw`) and will be interpreted as a (possibly fractional) number of bytes.
Description Maximum allowed size of frames for this host (in bytes:bits), used as input value by the scheduler, if the attribute `max_frame_size_soft` is not set. Note that the default value is not the absolute maximum. Larger frames can be specified, if additional bandwidth is needed.
If `use_x_frames` is specified, the `max_frame_size` includes the C-state transported by X-frames.

➤ Name `max_frame_size_soft`
Type `Bit_Size`
Default `automatically computed`
Syntax Specified as `[bytes][:bits]`.
Examples:

- 2 = two bytes
- 1.5 = 12 bits
- :1 = one bit
- 1:2 = 10 bits

If you set this attribute via a script, and pass an integer to the `set` function (e.g., `raw=0`), you must specify a length in bits.
Note: regular expressions are allowed in raw values (i.e., in the interactive editor and in scripts calling `set_raw`) and will be interpreted as a (possibly fractional) number of bytes.
Description Maximum allocation target (in bytes:bits) for N-frames of this host. The scheduler will use `max_frame_size_soft`, but N-frames exceeding `max_frame_size_soft` (but not exceeding `max_frame_size`) will not be considered as erroneous. Use this parameter to nudge the scheduler if you get `overfull frame` errors although there seems to be enough space for all messages.

➤ Name `min_frame_size`
Type `Bit_Size`
Default `automatically computed`
Syntax Specified as `[bytes][:bits]`.
Examples:

- 2 = two bytes
- 1.5 = 12 bits
- :1 = one bit
- 1:2 = 10 bits

If you set this attribute via a script, and pass an integer to the `set` function (e.g., `raw=0`), you must specify a length in bits.
Note: regular expressions are allowed in raw values (i.e., in the interactive editor and in scripts calling `set_raw`) and will be interpreted as a (possibly fractional) number of bytes.
Description Minimum size (in bytes:bits) reserved for frames (can be used to reserve bandwidth for this host).
This includes the C-state transported by X-Frames if `use_x_frames` is specified.

- Name `allow_schedule_extension`
  Type `Boolean`
  Default `no`
  Syntax The following string values are accepted as valid Boolean values: 0, 1, f, false, n, no, t, true, y, yes
  Description If true, the scheduler will include this host in the next schedule step. Once set to false, the host cannot participate in any further schedule steps.

`TTA.Host_in_Cluster`: optional attributes

Scope: TTP Plan scope

- Name `allow_coldstart`
  Type `Boolean`
  Default `automatically computed`
  Syntax The following string values are accepted as valid Boolean values: 0, 1, f, false, n, no, t, true, y, yes
  Description Allows the controller to initiate a the coldstart procedure. Usually all controllers can be allowed to coldstart, unless specific application design rules dictate otherwise. The permission may be revoked (by setting this attribute to 'No') if a controller is not allowed to perform a coldstart for protocol reasons

- Name `allow_coldstart_integration`
  Type `Boolean`
  Default `yes`
  Syntax The following string values are accepted as valid Boolean values: 0, 1, f, false, n, no, t, true, y, yes
  Description Allows the controller to integrate in the coldstart procedure.

- Name `external_rate_correction_allowed`
  Type `Boolean`
  Default `no`
  Syntax The following string values are accepted as valid Boolean values: 0, 1, f, false, n, no, t, true, y, yes
  Description If set to 'yes', this host is allowed to influence the reference time. This feature is mainly used for inter-cluster synchronization and for eliminating cluster drift by means of a GPS receiver or a similar external time source. This is not available in all protocols.

- Name `is_foreign`
  Type `Boolean`
  Default `no`
  Syntax The following string values are accepted as valid Boolean values: 0, 1, f, false, n, no, t, true, y, yes
  Description A foreign node is not synchronized with the cluster; for example, if it acts as a gateway.

➤ Name `is_optional`
Type `Boolean`
Default `no`
Syntax The following string values are accepted as valid Boolean values: 0, 1, f, false, n, no, t, true, y, yes
Description This parameter is used to determine the actively synchronizing nodes of the cluster, i.e., those nodes actively contributing to the clock synchronization in the cluster; it may only be set to "no" if this host is available in all configurations of the cluster.
The effect of the set value is on the computation of the clock synchronization parameters: an optional host is never part of the nodes contributing to the clock synchronization algorithm. A typical use for an optional host is to reserve a slot for future expansion, or to specify hosts that may be missing even during normal cluster operation.

`TTA.Host_in_Web`: optional attributes

Scope: TTP Plan scope

➤ Name `c_state_round_spec`
Type `Round_Pattern`
Default `None`
Syntax The value specifies a transmission period and an optional phase.
The value of the phase must be between `1` and the specified period.
Examples:

- **`2` = every other round (starting in the first round),** i.e., round 1, 3, 5 ...
- **`2 from 2` = every other round (starting in the second round),** i.e., round 2, 4, 6 ...

Description Rounds in which the node shall transmit C-state information, i.e., I-frames or X-frames (depending on the attribute `use_x_frames`).

➤ Name `edge_jitter_tolerance_ns`
Type `Nano_Time_Span`
Default `automatically computed`
Syntax The default unit is ns. If you specify another unit, it must be separated from the number by at least one space. You can use the following units: ms, ns, ps, s, us.
Description The edge jitter tolerance specifies the tolerance of the decoder against jitter for MFM and Manchester encoding (in nanoseconds). Large values lead to a robust data connection. Smaller values may be used to test the physical transmission layer for timing problems. Set `edge_jitter_tolerance` to a multiple of `(1000 / controller_type.clock_MHz)`.
A transmission code like MFM uses the distance between signal transitions to encode the data. In a TTP transmission encoded with MFM that means the timing of the transitions from high to low and from low to high depends on the data. The shortest possible distance between any two transitions is one bitcell, the next possible distance is 1.5 bitcells, and the next possible distance is 2 bitcells from the previous transition. MFM code has only these three possibilities: next transition after 1.0, 1.5, or 2.0 bitcells.
A transition after only half a bitcell or after 3 bitcells is certainly invalid for MFM. But what about 1.2 bitcells?
Physically, timing is never exact, of course; there will be transitions at the receiver after for example 0.98, 1.01, 1.49, 1.52, 1.98 and 2.01 microseconds in a good and fault-free system. It is easy to 'guess' in these cases that these were proper transmissions which were just slightly skewed.
The `edge_jitter_tolerance` tells the receivers how strict they should be when a signal transition occurs 'about at the right time, but not quite', i.e., it defines if a transition that is observed, e.g., 1.4 bitcells after the previous transition at the receiver should be considered a fault or a 'skewed' transition at 1.5 bitcells. It could be a proper, but 'rather early' signal from the sender -- in this case we want to 'tolerate' it -- or it could be a truly faulty sender, or, more likely, an external disturbance (EMI) which has hit the frame. In the latter cases we normally want the error to be detected. In the strictest setting, the receiver will allow only a minimal deviation from the expected time; in the 'laxest' case, it will try to decode anything.
A 'high strictness' will most likely exclude noise that would otherwise result in a (probably incorrect) decoded bit, which would later have to be detected by the CRC; but it requires that the rising and falling edges are transmitted with very little falsification ('signal skew').
A 'high laxity' allows for more 'signal skew' between senders and receivers, but noise will more likely result in seemingly correct MFM patterns which then are found faulty by the CRC checking mechanism.

➤ Name `uses_channels`
Type `Web_Channel_List`
Default `automatically computed`
Description Specifies the channels to which the host is connected.

`TTA.Host_uses_Slot`: optional attributes

Scope: TTP Plan scope

➤ Name `mux_period`
Type `Int`
Default `1`
Description Specifies multiplex period in rounds. It has to be a valid period in the cluster, where 1 means that the host uses its slot unmultiplexed.

➤ Name `mux_round`
Type `Int`
Default `1`
Description Specifies the first round in which the host uses the slot.

`TTA.MUX_Ghost`: optional attributes

Scope: TTP Plan scope

➤ Name `SYF`
Type `Boolean`
Default `automatically computed`
Syntax The following string values are accepted as valid Boolean values: 0, 1, f, false, n, no, t, true, y, yes
Description If set to “yes”, this host is one of the actively synchronizing nodes. Note that, in this case, `allow_active_role` must also be set to “yes” and the `Host_in_Cluster` attribute `is_optional` must be set to “no”.

➤ Name `allow_mode_change`
Type `Boolean`
Default `yes`
Syntax The following string values are accepted as valid Boolean values: 0, 1, f, false, n, no, t, true, y, yes
Description If true, this host is allowed to issue a cluster-mode change request.

➤ Name `min_integration_count`
Type `Int`
Default `2`
Description To disable the communication blackout for one node, the `minimum_integration_count` (MIC) must be set to 1 (at least for some nodes). This is necessary because of the scenario where one node (X- or I-frame sender) is the only active node (and does not shut down because of the disabled communication blackout detection). In this case, only nodes with MIC=1 can integrate into the running cluster.
Plugin This attribute is only available for certain plugins

➤ Name `perform_clique_detection`
Type `Boolean`
Default `True`
Syntax The following string values are accepted as valid Boolean values: 0, 1, f, false, n, no, t, true, y, yes
Description In order to optimize operation of the M-Hub, this attribute allows to selectively disable clique and communication blackout detection for a node. This means it is possible to clear the PC (perform clique detection) flag in each round-slot entry. Usually it is set in the node's own sending slot.
Plugin This attribute is only available for certain plugins

`TTA.Message`: optional attributes

Scope: TTP Plan scope

➤ Name `agreement`
Type `RD_Agreement`
Default `automatically computed`
Description Replica-deterministic agreement algorithm.
This parameter is only necessary for non-replica-deterministic messages sent by replicated subsystems.

➤ Name `caption`
Type `Text`
Default `automatically computed`
Description Caption displayed by TTP-View instruments for objects of this type.

➤ Name `is_h_state`
Type `Boolean`
Default `no`
Syntax The following string values are accepted as valid Boolean values: 0, 1, f, false, n, no, t, true, y, yes
Description Specifies if the message carries the h-state of a replicated task.

- Name `max_prd`
  Type `Micro_Time_Span`
  Default `None`
  Syntax The default unit is us. If you specify another unit, it must be separated from the number by at least one space. You can use the following units: ms, ns, ps, s, us.
  Description Maximum post-receive duration (in microseconds): specifies the maximum duration between the end of transmission and the consumption of the message.

- Name `max_psd`
  Type `Micro_Time_Span`
  Default `None`
  Syntax The default unit is us. If you specify another unit, it must be separated from the number by at least one space. You can use the following units: ms, ns, ps, s, us.
  Description Maximum pre-send duration (in microseconds): specifies the maximum duration between the creation of the message and the start of transmission.

- Name `sender_status`
  Type `Boolean`
  Default `no`
  Syntax The following string values are accepted as valid Boolean values: 0, 1, f, false, n, no, t, true, y, yes
  Description Specifies if a sender-status bit is transmitted along with the message. A sender-status bit allows the sending host to tell the receivers of the message whether the message value is valid or not.

- Name `subsystem`
  Type `Link`
  Default `None`
  Description Subsystem that sends the message

- Name `validity_span`
  Type `Micro_Time_Span`
  Default `None`
  Syntax The default unit is us. If you specify another unit, it must be separated from the number by at least one space. You can use the following units: ms, ns, ps, s, us.
  Description Amount of time the message remains valid (microseconds).

`TTA.Message_in_Msg_Box`: optional attributes

Scope: TTP˙Plan scope

➤ Name `bool_usage`
Type `Boolean`
Default `automatically computed`
Syntax The following string values are accepted as valid Boolean values: 0, 1, f, false, n, no, t, true, y, yes
Description Specifies whether the value of this message is just used in Boolean context.
For 1-bit messages this is per default `yes`. This means that the message is only used in Boolean context which allows for some optimization in the generated code. If this is enabled, you cannot rely on getting the correct integer value for this message. If you're really interested in the integer value (0 or 1), you have to set this to `no`. For messages greater than 1 bit, this is `no` per default. It does not make much sense to set it to `yes` in such cases.

`TTA.Msg_Box`: optional attributes

Scope: TTP Plan scope

➤ Name `byte_order`
Type `Byte_Order`
Default `automatically computed`
Description Byte order used for the transmission of data in the `Msg_Box`.

➤ Name `caption`
Type `Text`
Default `automatically computed`
Description Caption displayed by TTP-View instruments for objects of this type.

➤ Name `checksum`
Type `Checksum`
Default `None`
Description Checksum to be used to check validity of data in frame.

➤ Name `sender_status`
Type `Boolean`
Default `no`
Syntax The following string values are accepted as valid Boolean values: 0, 1, f, false, n, no, t, true, y, yes
Description Specifies if a sender-status bit is transmitted along with the message. A sender-status bit allows the sending host to tell the receivers of the message whether the message value is valid or not.

➤ Name `subsystem`
Type `Link`
Default `None`
Description Subsystem that sends the message

`TTA.Msg_Type_A`: optional attributes

Scope: TTP˙Plan scope

➤ Name `agreement`
Type `RD_Agreement`
Default `automatically computed`
Description Replica-deterministic agreement algorithm.
Setting this parameter to any value except `RD_1_valid` defines all messages associated to this type as non-replica-deterministic (this can be overridden by setting the `agreement` attribute of the message to `RD_1_valid`).

`TTA.Msg_Type_A`: optional attributes

Scope: TTP˙Plan framework scope

➤ Name `agreement`
Type `RD_Agreement`
Default `automatically computed`
Description Replica-deterministic agreement algorithm.
Setting this parameter to any value except `RD_1_valid` defines all messages associated to this type as non-replica-deterministic (this can be overridden by setting the `agreement` attribute of the message to `RD_1_valid`).

`TTA.Msg_Type_P`: optional attributes

Scope: TTP˙Plan scope

➤ Name `agreement`
Type `RD_Agreement`
Default `automatically computed`
Description Replica-deterministic agreement algorithm.
Setting this parameter to any value except `RD_1_valid` defines all messages associated to this type as non-replica-deterministic (this can be overridden by setting the `agreement` attribute of the message to `RD_1_valid`).

➤ Name `type_length`
Type `Bit_Size`
Default `automatically computed`
Syntax Specified as `[bytes][:bits]`.
Examples:

- 2 = two bytes
- 1.5 = 12 bits
- :1 = one bit
- 1:2 = 10 bits

If you set this attribute via a script, and pass an integer to the `set` function (e.g., `raw=0`), you must specify a length in bits.
Note: regular expressions are allowed in raw values (i.e., in the interactive editor and in scripts calling `set_raw`) and will be interpreted as a (possibly fractional) number of bytes.
Description Length of the message type in bytes:bits (i.e., what `sizeof (<type>)` returns).
This attribute specifies the length of a message of this message type on the node. This might be larger than the actual `length` of the message itself because, for example, a message of type `byte` will always be 8 bits on the node, even if its `length` is only 6 bits.
Caution: the internal unit for this attribute is bits (because all other lengths are measured in bits, too). Nevertheless, the value must denote a valid size in bytes (i.e., valid for the target hardware).

➤ Name `typedef`
Type `Text`
Default `None`
Description C-type definition for this type.
The type definition in the generated C code must be known to the compiler, otherwise a `mismatch` error will occur. Therefore the `typedef` has to match the specified `type_cat` and `type_length`. This especially applies to user-defined type definitions.

➤ Name `limit_high`
Type `Float`
Default `automatically computed`
Description Upper alarm limit for visualization. Values above this limit are displayed in red.

➤ Name `limit_low`
Type `Float`
Default `automatically computed`
Description Lower alarm limit for visualization. Values below this limit are displayed in red.

- Name `major_ticks`
  Type `Int`
  Default `automatically computed`
  Description Distance between major tick marks for visualization.

- Name `minor_ticks`
  Type `Int`
  Default `automatically computed`
  Description Number of minor ticks between two major ticks for visualization.

- Name `offset`
  Type `Float`
  Default `automatically computed`
  Description Offset for linear transformation of message values to the physical domain: `offset + <value> * slope`.

- Name `scale_high`
  Type `Float`
  Default `automatically computed`
  Description Upper end of scale for visualization. The value of `scale_high` must be greater than or equal to the maximum physical value.

- Name `scale_low`
  Type `Float`
  Default `automatically computed`
  Description Lower end of scale for visualization. The value of `scale_low` must be smaller than or equal to the minimum physical value.

- Name `slope`
  Type `Float`
  Default `automatically computed`
  Description Factor for linear transformation of message values to the physical domain: `offset + <value> * slope`.

- Name `unit`
  Type `String`
  Default `automatically computed`
  Description Unit of measurement for the objects of this type, e.g., km/h or rpm.

- Name `view_instrument`
  Type `TTPview-Instrument`
  Default `automatically computed`
  Description Default instrument used by TTP-View to display messages of this type.

`TTA.Msg_Type_P`: optional attributes

Scope: TTP-Plan framework scope

- Name `agreement`
  Type `RD_Agreement`
  Default `automatically computed`
  Description Replica-deterministic agreement algorithm.
  Setting this parameter to any value except `RD_1_valid` defines all messages associated to this type as non-replica-deterministic (this can be overridden by setting the `agreement` attribute of the message to `RD_1_valid`).
- Name `limit_high`
  Type `Float`
  Default `automatically computed`
  Description Upper alarm limit for visualization. Values above this limit are displayed in red.
- Name `limit_low`
  Type `Float`
  Default `automatically computed`
  Description Lower alarm limit for visualization. Values below this limit are displayed in red.
- Name `major_ticks`
  Type `Int`
  Default `automatically computed`
  Description Distance between major tick marks for visualization.
- Name `minor_ticks`
  Type `Int`
  Default `automatically computed`
  Description Number of minor ticks between two major ticks for visualization.
- Name `offset`
  Type `Float`
  Default `automatically computed`
  Description Offset for linear transformation of message values to the physical domain: `offset + <value> * slope`.
- Name `scale_high`
  Type `Float`
  Default `automatically computed`
  Description Upper end of scale for visualization. The value of `scale_high` must be greater than or equal to the maximum physical value.
- Name `scale_low`
  Type `Float`
  Default `automatically computed`
  Description Lower end of scale for visualization. The value of `scale_low` must be smaller than or equal to the minimum physical value.

➤ Name `slope`
Type `Float`
Default `automatically computed`
Description Factor for linear transformation of message values to the physical domain: `offset + <value> * slope`.

➤ Name `unit`
Type `String`
Default `automatically computed`
Description Unit of measurement for the objects of this type, e.g., km/h or rpm.

➤ Name `view_instrument`
Type `TTPview-Instrument`
Default `automatically computed`
Description Default instrument used by TTP-View to display messages of this type.

`TTA.Msg_Type_S`: optional attributes

Scope: TTP˙Plan scope

➤ Name `agreement`
Type `RD_Agreement`
Default `automatically computed`
Description Replica-deterministic agreement algorithm.
Setting this parameter to any value except `RD_1_valid` defines all messages associated to this type as non-replica-deterministic (this can be overridden by setting the `agreement` attribute of the message to `RD_1_valid`).

`TTA.Msg_Type_S`: optional attributes

Scope: TTP˙Plan framework scope

➤ Name `agreement`
Type `RD_Agreement`
Default `automatically computed`
Description Replica-deterministic agreement algorithm.
Setting this parameter to any value except `RD_1_valid` defines all messages associated to this type as non-replica-deterministic (this can be overridden by setting the `agreement` attribute of the message to `RD_1_valid`).

`TTA.RDA`: optional attributes

Scope: TTP˙Plan framework scope

➤ Name `handles_type_cat`
Type `Type-Cat-Selection`
Default `automatically computed`
Description Specifies which type-cats can be handled by this agreement

`TTA.Sched_Step_CS`: optional attributes

Scope: TTP Plan scope

➤ Name `agreement`
Type `RD_Agreement`
Default `automatically computed`
Description Replica-deterministic agreement algorithm.
This parameter is only necessary for non-replica-deterministic messages sent by replicated subsystems.

➤ Name `caption`
Type `Text`
Default `automatically computed`
Description Caption displayed by TTP-View instruments for objects of this type.

➤ Name `is_h_state`
Type `Boolean`
Default `no`
Syntax The following string values are accepted as valid Boolean values: 0, 1, f, false, n, no, t, true, y, yes
Description Specifies if the message carries the h-state of a replicated task.

➤ Name `max_prd`
Type `Micro_Time_Span`
Default `None`
Syntax The default unit is us. If you specify another unit, it must be separated from the number by at least one space. You can use the following units: ms, ns, ps, s, us.
Description Maximum post-receive duration (in microseconds): specifies the maximum duration between the end of transmission and the consumption of the message.

➤ Name `max_psd`
Type `Micro_Time_Span`
Default `None`
Syntax The default unit is us. If you specify another unit, it must be separated from the number by at least one space. You can use the following units: ms, ns, ps, s, us.
Description Maximum pre-send duration (in microseconds): specifies the maximum duration between the creation of the message and the start of transmission.

➤ Name `sender_status`
Type `Boolean`
Default `no`
Syntax The following string values are accepted as valid Boolean values: 0, 1, f, false, n, no, t, true, y, yes
Description Specifies if a sender-status bit is transmitted along with the message. A sender-status bit allows the sending host to tell the receivers of the message whether the message value is valid or not.

➤ Name `subsystem`
Type `Link`
Default `None`
Description Subsystem that sends the message

➤ Name `validity_span`
Type `Micro_Time_Span`
Default `None`
Syntax The default unit is us. If you specify another unit, it must be separated from the number by at least one space. You can use the following units: ms, ns, ps, s, us.
Description Amount of time the message remains valid (microseconds).

`TTA.Slot`: optional attributes

Scope: TTP˙Plan scope

➤ Name `min_duration`
Type `Micro_Time_Span`
Default `0`
Syntax The default unit is us. If you specify another unit, it must be separated from the number by at least one space. You can use the following units: ms, ns, ps, s, us.
Description Minimum duration of slot (in microseconds).

➤ Name `sort_key`
Type `Int`
Default `automatically computed`
Description Sort key for slot allocation -- slots with lower keys get lower round-phases (slots with equal `sort_key` are put in random order).

`TTA.Subsystem`: optional attributes

Scope: TTP˙Plan scope

- Name `permanence_delay`
  Type `Int`
  Default `0`
  Description Specifies the number of non-optional slots that must pass before a message of this subsystem is used by the receiver.
  To achieve full replica determinism, a message replica can be used by the receiver only when the sending node has been acknowledged by TTP. This can happen not earlier than 2 non-optional slots after the host's sending slot and not later than 1 round after the host's sending slot.
  0 - means no replica determinism in case of a transmission failure.
  2 - means replica determinism in case of exactly one transmission failure.
  2 + n - means replica determinism in case of a transmission failure and at most n additional failures of other hosts.
  Please note : The higher this value is set, the more failures are tolerated while still having replica determinism, but on the other hand, task scheduling becomes harder.

- Name `reintegration_timeout`
  Type `Int`
  Default `2`
  Description Only used if `reintegration_type` is `h_state_reintegration`. Specifies how many rounds this subsystems waits for a valid h-state before it starts up using the initial h-state.

- Name `reintegration_type`
  Type `Reintegration_Type`
  Default `Reinit_Reintegration`
  Description Specifies the type of reintegration required for this subsystem.

- Name `startup_quota`
  Type `Int`
  Default `None`
  Description Specifies the number of subsystems which must be active before the mode change from the TTP startup mode to the TTP application mode can be requested.

- Name `status`
  Type `Boolean`
  Default `no`
  Syntax The following string values are accepted as valid Boolean values: 0, 1, f, false, n, no, t, true, y, yes
  Description Specifies if this subsystem needs a dedicated subsystem status message to indicate if the subsystem is running correctly.

➤ Name `status_r_period`
Type `Int`
Default `automatically computed`
Description Specifies the period of the associated status message (in rounds). An `r_period` of 1 means the message is sent every round whereas 2 means every second round, etc. (default: once per cluster cycle).

➤ Name `use_full_h_state_as_guard`
Type `Boolean`
Default `no`
Syntax The following string values are accepted as valid Boolean values: 0, 1, f, false, n, no, t, true, y, yes
Description Specifies whether all h-state messages sent by this subsystem should be used as guard for the execution of the tasks belonging to this subsystem.

`TTA.Subsystem_Status_Message`: optional attributes

Scope: TTP˙Plan scope

➤ Name `agreement`
Type `RD_Agreement`
Default `automatically computed`
Description Replica-deterministic agreement algorithm.
This parameter is only necessary for non-replica-deterministic messages sent by replicated subsystems.

➤ Name `caption`
Type `Text`
Default `automatically computed`
Description Caption displayed by TTP-View instruments for objects of this type.

➤ Name `is_h_state`
Type `Boolean`
Default `no`
Syntax The following string values are accepted as valid Boolean values: 0, 1, f, false, n, no, t, true, y, yes
Description Specifies if the message carries the h-state of a replicated task.

➤ Name `max_prd`
Type `Micro_Time_Span`
Default `None`
Syntax The default unit is us. If you specify another unit, it must be separated from the number by at least one space. You can use the following units: ms, ns, ps, s, us.
Description Maximum post-receive duration (in microseconds): specifies the maximum duration between the end of transmission and the consumption of the message.

➤ Name `max_psd`
Type `Micro_Time_Span`
Default `None`
Syntax The default unit is us. If you specify another unit, it must be separated from the number by at least one space. You can use the following units: ms, ns, ps, s, us.
Description Maximum pre-send duration (in microseconds): specifies the maximum duration between the creation of the message and the start of transmission.

➤ Name `sender_status`
Type `Boolean`
Default `no`
Syntax The following string values are accepted as valid Boolean values: 0, 1, f, false, n, no, t, true, y, yes
Description Specifies if a sender-status bit is transmitted along with the message. A sender-status bit allows the sending host to tell the receivers of the message whether the message value is valid or not.

➤ Name `subsystem`
Type `Link`
Default `None`
Description Subsystem that sends the message

➤ Name `validity_span`
Type `Micro_Time_Span`
Default `None`
Syntax The default unit is us. If you specify another unit, it must be separated from the number by at least one space. You can use the following units: ms, ns, ps, s, us.
Description Amount of time the message remains valid (microseconds).

`TTA.Virtual_Controller`: optional attributes

Scope: TTP Plan framework scope

➤ Name `ifg_ns`
Type `Nano_Time_Span`
Default `0`
Syntax The default unit is ns. If you specify another unit, it must be separated from the number by at least one space. You can use the following units: ms, ns, ps, s, us.
Description Minimum inter-frame gap required by communication controller (in nanoseconds).

➤ Name `max_slot_duration_MT`
Type `MT_Time_Span`
Default `TFL.Environment.practically_infinite`
Description Maximum duration of a slot (in macroticks).

➤ Name `may_mix_frame_types`
Type `Boolean`
Default `no`
Syntax The following string values are accepted as valid Boolean values: 0, 1, f, false, n, no, t, true, y, yes
Description Specifies if different frame types can be mixed in the same round-slot (i.e. on different channels).

➤ Name `protocol`
Type `Protocol`
Default `Protocol_V_0_6`
Description Protocol version used.

`TTA.Web`: optional attributes

Scope: TTP Plan scope

➤ Name `may_mix_frame_types`
Type `Boolean`
Default `yes`
Syntax The following string values are accepted as valid Boolean values: 0, 1, f, false, n, no, t, true, y, yes
Description Specifies if different frame types may be mixed in the same slot and round. Even if this flag is set to `yes`, frame types can only be mixed if the controller type used for the cluster supports this, i.e., the `may_mix_frame_types` attribute of the controller (chosen via `controller_type`) must be set to `yes` as well.

➤ Name `physical_interface`
Type `Physical_Interface`
Default `automatically computed`
Description Physical interface to be used for this web (overwrites the default `physical_interface` of the cluster).

➤ Name `use_single_cluster_mode`
Type `Boolean`
Default `no`
Syntax The following string values are accepted as valid Boolean values: 0, 1, f, false, n, no, t, true, y, yes
Description If set, the web uses only a single TTP mode, the `Working_Mode`.

`TTP.Clock_Sync_Multiweb`: optional attributes

Scope: TTP Plan framework scope

- Name `macro_tick_length`
  Type `Nano_Time_Span`
  Default `5000`
  Syntax The default unit is ns. If you specify another unit, it must be separated from the number by at least one space. You can use the following units: ms, ns, ps, s, us.
  Description Granularity of the global time (in nanoseconds).
  This value defines the duration of the global time tick (macrotick) generated by the controllers. It must be larger than the required precision in order to ensure a 'reasonable' time base for the system.

- Name `precision`
  Type `Nano_Time_Span`
  Default `4800`
  Syntax The default unit is ns. If you specify another unit, it must be separated from the number by at least one space. You can use the following units: ms, ns, ps, s, us.
  Description Limit for the skew of the communication controller clocks (in nanoseconds).
  All controllers periodically (i.e., in each resynchronization interval) check whether their local clock is synchronized to the rest of the cluster (fault-tolerant average) within +/- half the stated precision. If that is not the case, they raise a synchronization protocol error. This allows the application to rely on a global time base which is guaranteed to be synchronized in the whole cluster, with no more skew than the specified precision.

- Name `resync_interval_slots`
  Type `Int`
  Default `automatically computed`
  Description Number of 'active' round-slots in the resynchronization interval. The value `0` (zero) means 'all active slots'. A number of 4 means that resynchronization is performed after every four round-slots of actively synchronizing nodes.
  This parameter controls how many nodes with the 'good oscillators' define one resynchronization interval. If no value is specified, all active slots form one synchronization interval, giving an interval of one round. This may be too long for large clusters.

➤ Name `resync_rate`
Type `Int`
Default `25`
Description Specifies the rate at which the clock correction term is applied to the local clock (in nanoseconds per macrotick).
This parameter defines the 'average' maximum amount of time by which the macroticks can become longer or shorter while the communication controller applies the clock state correction term during clock synchronization in each resynchronization interval. A large `resync_rate` allows the nodes to become synchronized fast after performing the clock synchronization algorithm, a small value gives a smoother macrotick time base. Usually this rate should be no larger than 10% of the macrotick, but it can be much smaller.
This parameter can be smaller than (or not a multiple of) a controller's oscillator tick; as a macrotick may be prolonged/shortened only by the oscillator tick length, this parameter only describes the 'average' modification of the macrotick length during synchronization, and is actually implemented as 'free running macroticks'.
Regardless of this parameter, the total amount of change per resynchronization interval is limited by half the `precision` parameter.

➤ Name `web_sync_granularity_r`
Type `Int`
Default `1`
Description Granularity in rounds the web sync algorithm assures to be bound within `max_web_deviation`.

`TTP.Clock_Sync_Standard`: optional attributes

Scope: TTP Plan framework scope

➤ Name `macro_tick_length`
Type `Nano_Time_Span`
Default `5000`
Syntax The default unit is ns. If you specify another unit, it must be separated from the number by at least one space. You can use the following units: ms, ns, ps, s, us.
Description Granularity of the global time (in nanoseconds).
This value defines the duration of the global time tick (macrotick) generated by the controllers. It must be larger than the required precision in order to ensure a 'reasonable' time base for the system.

➤ Name `precision`
Type `Nano_Time_Span`
Default `4800`
Syntax The default unit is ns. If you specify another unit, it must be separated from the number by at least one space. You can use the following units: ms, ns, ps, s, us.
Description Limit for the skew of the communication controller clocks (in nanoseconds).
All controllers periodically (i.e., in each resynchronization interval) check whether their local clock is synchronized to the rest of the cluster (fault-tolerant average) within +/- half the stated precision. If that is not the case, they raise a synchronization protocol error. This allows the application to rely on a global time base which is guaranteed to be synchronized in the whole cluster, with no more skew than the specified precision.

➤ Name `resync_interval_slots`
Type `Int`
Default `automatically computed`
Description Number of 'active' round-slots in the resynchronization interval. The value `0` (zero) means 'all active slots'. A number of 4 means that resynchronization is performed after every four round-slots of actively synchronizing nodes.
This parameter controls how many nodes with the 'good oscillators' define one resynchronization interval. If no value is specified, all active slots form one synchronization interval, giving an interval of one round. This may be too long for large clusters.

➤ Name `resync_rate`
Type `Int`
Default `25`
Description Specifies the rate at which the clock correction term is applied to the local clock (in nanoseconds per macrotick).
This parameter defines the 'average' maximum amount of time by which the macroticks can become longer or shorter while the communication controller applies the clock state correction term during clock synchronization in each resynchronization interval. A large `resync_rate` allows the nodes to become synchronized fast after performing the clock synchronization algorithm, a small value gives a smoother macrotick time base. Usually this rate should be no larger than 10% of the macrotick, but it can be much smaller.
This parameter can be smaller than (or not a multiple of) a controller's oscillator tick; as a macrotick may be prolonged/shortened only by the oscillator tick length, this parameter only describes the 'average' modification of the macrotick length during synchronization, and is actually implemented as 'free running macroticks'.
Regardless of this parameter, the total amount of change per resynchronization interval is limited by half the `precision` parameter.

`TTP.Controller_T3C2NF`: optional attributes

Scope: TTP Plan framework scope

➤ Name `asynchronous_preamble_cutoff_ns`
Type `Nano_Time_Span`
Default `501`
Syntax The default unit is ns. If you specify another unit, it must be separated from the number by at least one space. You can use the following units: ms, ns, ps, s, us.
Description Parameter (in nanoseconds) to compensate for Manchester bug. See Application Note 136.

➤ Name `clock_MHz`
Type `Float`
Default `40.0`
Description Oscillator clock frequency in MHz.

➤ Name `function_LED0`
Type `Int`
Default `0`
Description Defines the function of LED-Pin 0.
0: Pin is general purpose output.
1: Pin reflects the corresponding TCU function.
2: Pin reflects the corresponding BG function.
3: Pin output is always ’1’.

➤ Name `function_LED1`
Type `Int`
Default `0`
Description Defines the function of LED-Pin 1.
0: Pin is general purpose output.
1: Pin reflects the corresponding TCU function.
2: Pin reflects the corresponding BG function.
3: Pin output is always ’1’.

➤ Name `function_LED2`
Type `Int`
Default `0`
Description Defines the function of LED-Pin 2. 0: Pin is general purpose output.
1: Pin reflects the corresponding TCU function.
2: Pin reflects the corresponding BG function.
3: Pin output is always ’1’.

➤ Name `input_output_LED0`
Type `Boolean`
Default `no`
Syntax The following string values are accepted as valid Boolean values:
0, 1, f, false, n, no, t, true, y, yes
Description Defines the LED-Pin 0 as output [no] or input/output [yes].

➤ Name `input_output_LED1`
Type `Boolean`
Default `no`
Syntax The following string values are accepted as valid Boolean values:
0, 1, f, false, n, no, t, true, y, yes
Description Defines the LED-Pin 1 as output [no] or input/output [yes].

➤ Name `input_output_LED2`
Type `Boolean`
Default `no`
Syntax The following string values are accepted as valid Boolean values:
0, 1, f, false, n, no, t, true, y, yes
Description Defines the LED-Pin 2 as output [no] or input/output [yes].

➤ Name `medl_text`
Type `String`
Default `None`
Description Free text that will be included in the MEDL. Depending on the type of communication controller and other information in the MEDL, this text may be truncated.
The `medl_text` is added to the MEDL identifier table and is then transferred to the generated code together with the MEDL.

➤ Name `protocol_LED`
Type `Boolean`
Default `no`
Syntax The following string values are accepted as valid Boolean values: 0, 1, f, false, n, no, t, true, y, yes
Description Defines the LED used for protocol activity in firmware version 2.*. If the flag is set, the protocol activity LED will be on LED pin 2 and the RPV on LED pin 0, if the flag is cleared, the protocol LED will be on LED pin 0 and the RPV on LED pin 2.

`TTP.Host_on_MHub`: optional attributes

Scope: TTP Plan scope

➤ Name `branch`
Type `Int`
Default `automatically computed`
Description M-Hub branch/port the host is connected to. Default is set to zero, when `TTA.Host` is sending in first `TTA.Slot`, otherwise to one.

➤ Name `velocity_of_propagation`
Type `Float`
Default `0.78`
Description Velocity of propagation (in percent).

`TTP.MHub`: optional attributes

Scope: TTP Plan scope

➤ Name `hold_off_time`
Type `Micro_Time_Span`
Default `15`
Syntax The default unit is us. If you specify another unit, it must be separated from the number by at least one space. You can use the following units: ms, ns, ps, s, us.
Description The minimum time (in us) before a frame can be received. The M-Hub does not accept any incoming data during this time, the acceptance window starts after this time has elapsed.

➤ Name `precision`
Type `Micro_Time_Span`
Default `automatically computed`
Syntax The default unit is us. If you specify another unit, it must be separated from the number by at least one space. You can use the following units: ms, ns, ps, s, us.
Description Precision of the MHUB (in us). The cluster's precision is taken as default value.

`TTP.Monitor_Host_TTPpowernode_C2NF`: optional attributes

Scope: TTP Plan framework scope

➤ Name `allow_active_role`
Type `Boolean`
Default `no`
Syntax The following string values are accepted as valid Boolean values: 0, 1, f, false, n, no, t, true, y, yes
Description If true, this host is allowed to send. Otherwise it can only receive.

➤ Name `rpv_startup_logic`
Type `Boolean`
Default `no`
Syntax The following string values are accepted as valid Boolean values: 0, 1, f, false, n, no, t, true, y, yes
Description Indicates whether the RPV pin should initially (i.e., until the first decision point) be set to high (`yes`) or low (`no`).

➤ Name `use_x_frames`
Type `Boolean`
Default `no`
Syntax The following string values are accepted as valid Boolean values: 0, 1, f, false, n, no, t, true, y, yes
Description Specifies whether the host should use X-frames instead of I-frames to transmit C-state information.

➤ Name `max_frame_size`
Type `Bit_Size`
Default `automatically computed`
Syntax Specified as `[bytes][:bits]`.
Examples:

- 2 = two bytes
- 1.5 = 12 bits
- :1 = one bit
- 1:2 = 10 bits

If you set this attribute via a script, and pass an integer to the `set` function (e.g., `raw=0`), you must specify a length in bits.
Note: regular expressions are allowed in raw values (i.e., in the interactive editor and in scripts calling `set_raw`) and will be interpreted as a (possibly fractional) number of bytes.
Description Maximum allowed size of frames for this host (in bytes:bits), used as input value by the scheduler, if the attribute `max_frame_size_soft` is not set. Note that the default value is not the absolute maximum. Larger frames can be specified, if additional bandwidth is needed.

➤ Name `max_frame_size_soft`
Type `Bit_Size`
Default `automatically computed`
Syntax Specified as `[bytes][:bits]`.
Examples:

- 2 = two bytes
- 1.5 = 12 bits
- :1 = one bit
- 1:2 = 10 bits

If you set this attribute via a script, and pass an integer to the `set` function (e.g., `raw=0`), you must specify a length in bits.
Note: regular expressions are allowed in raw values (i.e., in the interactive editor and in scripts calling `set_raw`) and will be interpreted as a (possibly fractional) number of bytes.
Description Maximum allocation target (in bytes:bits) for N-frames of this host. The scheduler will use `max_frame_size_soft`, but N-frames exceeding `max_frame_size_soft` (but not exceeding `max_frame_size`) will not be considered as erroneous. Use this parameter to nudge the scheduler if you get `overfull frame` errors although there seems to be enough space for all messages.

➤ Name `min_frame_size`
Type `Bit_Size`
Default `automatically computed`
Syntax Specified as `[bytes][:bits]`.
Examples:

- 2 = two bytes
- 1.5 = 12 bits
- :1 = one bit
- 1:2 = 10 bits

If you set this attribute via a script, and pass an integer to the `set` function (e.g., `raw=0`), you must specify a length in bits.
Note: regular expressions are allowed in raw values (i.e., in the interactive editor and in scripts calling `set_raw`) and will be interpreted as a (possibly fractional) number of bytes.
Description Minimum size (in bytes:bits) reserved for frames (can be used to reserve bandwidth for this host).

➤ Name `allow_schedule_extension`
Type `Boolean`
Default `no`
Syntax The following string values are accepted as valid Boolean values: 0, 1, f, false, n, no, t, true, y, yes
Description If true, the scheduler will include this host in the next schedule step. Once set to false, the host cannot participate in any further schedule steps.

`TTP.Protocol_V_0_6`: optional attributes

Scope: TTP Plan framework scope

➤ Name `coldstart_attempts`
Type `Int`
Default `5`
Description Maximum number of cold-start attempts a controller performs (0 = never stop).

➤ Name `max_membership_failure`
Type `Int`
Default `10`
Description Maximum number of successive membership failures before shutdown of node.

➤ Name `number_of_rounds_for_listen_timeout`
Type `Int`
Default `2`
Description Defines the number of rounds the controller will wait for an explicit C-state until it sends a coldstart-frame.

➤ Name `min_sync_hosts`
Type `Int`
Default `4`
Description Minimum required number of actively synchronizing nodes.

➤ Name `min_nodes`
Type `Int`
Default `4`
Description Minimum number of nodes in cluster.

12.3. Output Data

`ESC.Physical_Interface_MFM`: output attributes

Scope: TTP Plan framework scope

➤ Name `ox`
Type `Int`
Description Object index. For each object type, `ox` is a number unique for each instance between `0` and `<Type.count> - 1`.

`ESC.Physical_Interface_MII`: output attributes

Scope: TTP Plan framework scope

➤ Name `ox`
Type `Int`
Description Object index. For each object type, `ox` is a number unique for each instance between `0` and `<Type.count> - 1`.

`ESC.Physical_Interface_Manchester`: output attributes

Scope: TTP Plan framework scope

➤ Name `ox`
Type `Int`
Description Object index. For each object type, `ox` is a number unique for each instance between `0` and `<Type.count> - 1`.

`RPV.Group`: output attributes

Scope: TTP Plan scope

➤ Name `ox`
Type `Int`
Description Object index. For each object type, `ox` is a number unique for each instance between `0` and `<Type.count> - 1`.

RPV.Group_controls_Host: output attributes

Scope: TTP˙Plan scope

➤ Name `bit_number`
Type `Int`
Description Number of bit controlling the host. This denotes the position of the flag within the `N_Frame`.

RPV.Message: output attributes

Scope: TTP˙Plan scope

➤ Name `bit_number`
Type `Int`
Description Number of bit controlling the host. This denotes the position of the flag within the `N_Frame`.

➤ Name `ox`
Type `Int`
Description Object index. For each object type, `ox` is a number unique for each instance between `0` and `<Type.count> - 1`.

TTA.Channel: output attributes

Scope: TTP˙Plan scope

➤ Name `abbr`
Type `Name`
Description Short name used for derived object names and by the schedule editor.

➤ Name `number`
Type `Int`
Description The number of this channel.

➤ Name `ox`
Type `Int`
Description Object index. For each object type, `ox` is a number unique for each instance between `0` and `<Type.count> - 1`.

TTA.Checksum: output attributes

Scope: TTP˙Plan scope

➤ Name `ox`
Type `Int`
Description Object index. For each object type, `ox` is a number unique for each instance between `0` and `<Type.count> - 1`.

`TTA.Cluster`: output attributes

Scope: TTP˙Plan scope

- Name `frozen_schedule_cs`
  Type `Long_List`
  Description Checksums of all frozen schedule steps.
- Name `frozen_schedule_step`
  Type `Int`
  Description Indicates the last frozen schedule step. Once a schedule step is frozen, all commands affecting the schedule (e.g., `Make new schedule`, `Delete schedule`) leave the frozen schedule step alone and add or remove the next step.
- Name `ox`
  Type `Int`
  Description Object index. For each object type, `ox` is a number unique for each instance between `0` and `<Type.count> - 1`.
- Name `rounds`
  Type `Int`
  Description Number of rounds scheduled.
- Name `schedule_cs`
  Type `Long-Int`
  Description Checksum of last (unfrozen) schedule made.
- Name `schedule_errors`
  Type `Int`
  Description Number of schedule errors after scheduler finished.
- Name `schedule_step_cs_redundancy`
  Type `Int`
  Description Redundancy degree of the `schedule_step_id`.
- Name `scheduled_objects`
  Type `Int`
  Description Number of scheduled objects.
- Name `tc_period`
  Type `Float`
  Description Transmission cycle period, i.e., duration of the cluster cycle (in microseconds), rounded to multiples of the macrotick length.
- Name `latest_schedule_step`
  Type `Int`
  Description Number of current schedule step.

`TTA.Cluster_Mode`: output attributes

Scope: TTP˙Plan framework scope

- Name `bits_per_cycle`
  Type `Int`
  Description Bits transmitted per cluster cycle.
- Name `i_frames_per_cycle`
  Type `Int`
  Description Explicit C-state transmitted per cluster cycle.
- Name `instances_per_cycle`
  Type `Int`
  Description Message instances transmitted per cluster cycle.
- Name `messages_per_cycle`
  Type `Int`
  Description Message generations transmitted per cluster cycle.
- Name `mode_number`
  Type `Int`
  Description Number identifiying the cluster mode.
- Name `n_frames_per_cycle`
  Type `Int`
  Description N-frames transmitted per cluster cycle.
- Name `ox`
  Type `Int`
  Description Object index. For each object type, `ox` is a number unique for each instance between `0` and `<Type.count> - 1`.
- Name `round_duration_MT`
  Type `MT_Time_Span`
  Description Duration of round (in macroticks). For a correct schedule this must be equal to `Cluster.tr_duration`.
- Name `round_stretch_us`
  Type `Int`
  Description Stretch of round (in microseconds). This is the time added to the net transmission time and the inter-frame gaps to extend the `round_duration_us` to `Cluster.tr_period`.
- Name `rounds`
  Type `Int`
  Description Number of rounds scheduled.

`TTA.Cluster_uses_Message`: output attributes

Scope: TTP˙Plan scope

- Name `a_period`
  Type `Float`
  Description Actual transmission period of the message as computed by the scheduler (in microseconds).
- Name `a_periodicity`
  Type `Int`
  Description Base-2 logarithm of ratio between `a_period` and `TTA.Cluster.tr_period_r`. Defines the actual periodicity of the message, i.e., in which `tr_period` it is transmitted (every, every other, etc.).
- Name `a_phase`
  Type `Float`
  Description Actual (i.e., computed) transmission time for the first transmission of the message in the cluster cycle (in microseconds from cluster cycle start).
- Name `a_redundancy`
  Type `Int`
  Description Actual (scheduled) redundancy degree.
- Name `a_round`
  Type `Int`
  Description Earliest round where the message is actually transmitted.
- Name `a_rs_periods`
  Type `Float_List`
  Description List of delta times between round-slots carrying the message (in microseconds).
- Name `a_web_redundancy`
  Type `Int`
  Description Actual (scheduled) web redundancy degree, i.e., in which web(s) the message is actually transmitted.
- Name `transmission_duration`
  Type `Micro_Time_Span`
  Description Duration of message transmission (in microseconds).

`TTA.Controller_Set`: output attributes

Scope: TTP-Plan framework scope

- Name `ox`
  Type `Int`
  Description Object index. For each object type, `ox` is a number unique for each instance between `0` and `<Type.count> - 1`.

`TTA.H_State_Reintegration`: output attributes

Scope: TTP-Plan framework scope

➤ Name `ox`
Type `Int`
Description Object index. For each object type, `ox` is a number unique for each instance between `0` and `<Type.count> - 1`.

`TTA.Host`: output attributes

Scope: TTP-Plan scope

➤ Name `frames`
Type `Link_Set`
Description `Frames` linked to `Host`.

➤ Name `frozen_schedule_cs`
Type `Long_List`
Description Checksums of all frozen schedule steps.

➤ Name `frozen_schedule_step`
Type `Int`
Description Indicates the last frozen schedule step. Once a schedule step is frozen, all commands affecting the schedule (e.g., `Make new schedule`, `Delete schedule`) leave the frozen schedule step alone and add or remove the next step.

➤ Name `last_schedule_step`
Type `Int`
Description Indicates the latest schedule step this host participated in.

➤ Name `membershipbit`
Type `Int`
Description Bit number of this host in the membership vector.

➤ Name `ox`
Type `Int`
Description Object index. For each object type, `ox` is a number unique for each instance between `0` and `<Type.count> - 1`.

➤ Name `schedule_cs`
Type `Long-Int`
Description Checksum of last (unfrozen) schedule made.

➤ Name `schedule_errors`
Type `Int`
Description Number of schedule errors after scheduler finished.

➤ Name `scheduled_objects`
Type `Int`
Description Number of scheduled objects.

- Name `startup_timeout_MT`
  Type `MT_Time_Span`
  Description Unique amount of time (in macroticks) the controller waits until it sends the first cold-start frame.
- Name `sync_slot`
  Type `Int`
  Description Slot in which this host sends sync frames.
- Name `latest_schedule_step`
  Type `Int`
  Description Number of current schedule step.

`TTA.I_Frame`: output attributes

Scope: TTP˙Plan scope

- Name `c_state_size`
  Type `Bit_Size`
  Description Length of C-state data carried by frame (in bytes:bits).
- Name `channel`
  Type `Channel`
  Description Channel on which the frame is transmitted.
- Name `cni_address`
  Type `Hex`
  Description Offset of this frame to the CNI message area base address.
- Name `data_offset`
  Type `Int`
  Description Offset of Remote Pin Voting and message data in frame (in bits).
- Name `data_size`
  Type `Bit_Size`
  Description Length of message data (in bytes:bits). This includes any packing holes.
- Name `host`
  Type `Link`
  Description `Host` linked to `Frame`.
- Name `mode_number`
  Type `Int`
  Description `Cluster_Mode` the frame belongs to.
- Name `ox`
  Type `Int`
  Description Object index. For each object type, `ox` is a number unique for each instance between `0` and `<Type.count> - 1`.

- Name `r_slot`
  Type `Link`
  Description `R_slot` linked to `Frame`.
- Name `round`
  Type `Int`
  Description Number of round the frame is transmitted in.
- Name `rpv_size`
  Type `Bit_Size`
  Description Length of Remote Pin Voting data (in bytes:bits) carried by frame.
- Name `status_offset`
  Type `Int`
  Description Offset of status data in frame (in bytes).

`TTA.MUX_Ghost`: output attributes

Scope: TTP Plan scope

- Name `frames`
  Type `Link_Set`
  Description `Frames` linked to `Host`.
- Name `frozen_schedule_cs`
  Type `Long_List`
  Description Checksums of all frozen schedule steps.
- Name `frozen_schedule_step`
  Type `Int`
  Description Indicates the last frozen schedule step. Once a schedule step is frozen, all commands affecting the schedule (e.g., `Make new schedule`, `Delete schedule`) leave the frozen schedule step alone and add or remove the next step.
- Name `last_schedule_step`
  Type `Int`
  Description Indicates the latest schedule step this host participated in.
- Name `membershipbit`
  Type `Int`
  Description Bit number of this host in the membership vector.
- Name `ox`
  Type `Int`
  Description Object index. For each object type, `ox` is a number unique for each instance between `0` and `<Type.count> - 1`.
- Name `schedule_cs`
  Type `Long-Int`
  Description Checksum of last (unfrozen) schedule made.

➤ Name `schedule_errors`
Type `Int`
Description Number of schedule errors after scheduler finished.

➤ Name `scheduled_objects`
Type `Int`
Description Number of scheduled objects.

➤ Name `sync_slot`
Type `Int`
Description Slot in which this host sends sync frames.

➤ Name `latest_schedule_step`
Type `Int`
Description Number of current schedule step.

`TTA.Message`: output attributes

Scope: TTP Plan scope

➤ Name `ox`
Type `Int`
Description Object index. For each object type, `ox` is a number unique for each instance between `0` and `<Type.count> - 1`.

`TTA.Message_in_Msg_Box`: output attributes

Scope: TTP Plan scope

➤ Name `part_pos`
Type `Msg_Part_List`
Description Position (i.e., offset) and length of message parts (in bits) if the message extends over byte borders and is thus split into parts.

➤ Name `sender_status_pos`
Type `Msg_Part`
Description Position (i.e., offset) of sender status (in bits).

`TTA.Message_in_N_Frame`: output attributes

Scope: TTP Plan scope

➤ Name `box_offset`
Type `Int`
Description For messages inside a `Msg_Box`, the position of the `Msg_Box` in the frame.

➤ Name `length`
Type `Bit_Size`
Description Length of main body of `Message` (in bytes:bits) stored in `N_Frame` starting at `offset`.

➤ Name `length_f`
Type `Bit_Size`
Description Length of bit-rest of `Message` (in bytes:bits) stored in `N_Frame` starting at `offset_f`.

➤ Name `offset`
Type `Bit_Size`
Description Bit offset of main body of `Message` in `N_Frame` (in bytes:bits).

➤ Name `offset_f`
Type `Bit_Size`
Description Bit offset of bit-rest of `Message` in `N_Frame` (in bytes:bits).

➤ Name `part_pos`
Type `Msg_Part_List`
Description Position (i.e., offset) and length of message parts (in bits) if the message extends over byte borders and is thus split into parts.

➤ Name `sender_status_pos`
Type `Msg_Part`
Description Position (i.e., offset) of sender status (in bits).

`TTA.Msg_Box`: output attributes

Scope: TTP Plan scope

➤ Name `checksum_pos`
Type `Msg_Part`
Description Position of checksum in `Msg_Box`.

➤ Name `ox`
Type `Int`
Description Object index. For each object type, `ox` is a number unique for each instance between `0` and `<Type.count> - 1`.

`TTA.Msg_Type_A`: output attributes

Scope: TTP Plan scope

➤ Name `ox`
Type `Int`
Description Object index. For each object type, `ox` is a number unique for each instance between `0` and `<Type.count> - 1`.

➤ Name `type_cat`
Type `Type-Cat`
Description Category of type.
The `type_cat` specified must match the `type_cat_implied` as implied by the `typedef` attribute (if defined) of the `Msg_Type`.
INT = signed integer value,
UINT = unsigned integer value,
REAL = floating point value,
ARRAY = array type specified via `Msg_Type_A`,
STRUCT = struct type specified via `Msg_Type_S`.
This attribute defines the major type categories that are then subdivided into more specific categories. For example, the category INT contains types like `word`, `byte`, etc. Note: the tool might accept a user-defined `typedef`, but to avoid compiler errors, the type-related attributes have to match in any case!

`TTA.Msg_Type_A`: output attributes

Scope: TTP˙Plan framework scope

➤ Name `ox`
Type `Int`
Description Object index. For each object type, `ox` is a number unique for each instance between `0` and `<Type.count> - 1`.

`TTA.Msg_Type_P`: output attributes

Scope: TTP˙Plan scope

➤ Name `ox`
Type `Int`
Description Object index. For each object type, `ox` is a number unique for each instance between `0` and `<Type.count> - 1`.

`TTA.Msg_Type_P`: output attributes

Scope: TTP˙Plan framework scope

➤ Name `ox`
Type `Int`
Description Object index. For each object type, `ox` is a number unique for each instance between `0` and `<Type.count> - 1`.

`TTA.Msg_Type_S`: output attributes

Scope: TTP˙Plan scope

➤ Name `ox`
Type `Int`
Description Object index. For each object type, `ox` is a number unique for each instance between `0` and `<Type.count> - 1`.

➤ Name `type_cat`
Type `Type-Cat`
Description Category of type.
The `type_cat` specified must match the `type_cat_implied` as implied by the `typedef` attribute (if defined) of the `Msg_Type`.
INT = signed integer value,
UINT = unsigned integer value,
REAL = floating point value,
ARRAY = array type specified via `Msg_Type_A`,
STRUCT = struct type specified via `Msg_Type_S`.
This attribute defines the major type categories that are then subdivided into more specific categories. For example, the category INT contains types like `word`, `byte`, etc. Note: the tool might accept a user-defined `typedef`, but to avoid compiler errors, the type-related attributes have to match in any case!

`TTA.Msg_Type_S`: output attributes

Scope: TTP˙Plan framework scope

➤ Name `ox`
Type `Int`
Description Object index. For each object type, `ox` is a number unique for each instance between `0` and `<Type.count> - 1`.

`TTA.N_Frame`: output attributes

Scope: TTP˙Plan scope

➤ Name `c_state_size`
Type `Bit_Size`
Description Length of C-state data carried by frame (in bytes:bits).

➤ Name `channel`
Type `Channel`
Description Channel on which the frame is transmitted.

➤ Name `cni_address`
Type `Hex`
Description Offset of this frame to the CNI message area base address.

➤ Name `data_offset`
Type `Int`
Description Offset of Remote Pin Voting and message data in frame (in bits).

➤ Name `data_size`
Type `Bit_Size`
Description Length of message data (in bytes:bits). This includes any packing holes.

➤ Name `host`
Type `Link`
Description `Host` linked to `Frame`.

➤ Name `mode_number`
Type `Int`
Description `Cluster_Mode` the frame belongs to.

➤ Name `ox`
Type `Int`
Description Object index. For each object type, `ox` is a number unique for each instance between `0` and `<Type.count> - 1`.

➤ Name `r_slot`
Type `Link`
Description `R_slot` linked to `Frame`.

➤ Name `round`
Type `Int`
Description Number of round the frame is transmitted in.

➤ Name `rpv_size`
Type `Bit_Size`
Description Length of Remote Pin Voting data (in bytes:bits) carried by frame.

➤ Name `status_offset`
Type `Int`
Description Offset of status data in frame (in bytes).

`TTA.No_Reintegration`: output attributes

Scope: TTP Plan framework scope

➤ Name `ox`
Type `Int`
Description Object index. For each object type, `ox` is a number unique for each instance between `0` and `<Type.count> - 1`.

`TTA.RDA`: output attributes

Scope: TTP Plan framework scope

➤ Name `ox`
Type `Int`
Description Object index. For each object type, `ox` is a number unique for each instance between `0` and `<Type.count> - 1`.

`TTA.R_Slot`: output attributes

Scope: TTP˙Plan scope

- Name `SYF`
  Type `Boolean`
  Description If set to true, the host sending in this round-slot contributes to the clock synchronization.
- Name `clock_sync`
  Type `Boolean`
  Description If set to “yes”, the clock synchronization algorithm is performed in this round-slot.
- Name `cluster_mode`
  Type `Cluster_Mode`
  Description Cluster mode associated to this round-slot.
- Name `frames`
  Type `Link_Set`
  Description `Frames` linked to `R_slot`.
- Name `max_frame_size`
  Type `Bit_Size`
  Description Size of biggest frame transmitted in this slot (in bytes:bits).
- Name `max_valid_frame_size`
  Type `Bit_Size`
  Description Size of biggest valid frame transmitted in this slot (in bytes:bits).
- Name `net_duration`
  Type `MT_Time_Span`
  Description Time needed for the transmission of the largest frame transmitted in this slot (in macroticks).
- Name `number`
  Type `Int`
  Description MEDL position of this `R_Slot`.
- Name `ox`
  Type `Int`
  Description Object index. For each object type, `ox` is a number unique for each instance between `0` and `<Type.count> - 1`.
- Name `phase`
  Type `MT_Time_Span`
  Description Phase of round-slot relative to the cluster cycle (in macroticks).
- Name `round`
  Type `Int`
  Description Number of the round in which this round-slot is located.

➤ Name `rss_offset`
Type `Hex`
Description This offset has to be subtracted from the `cni_address` of frames of a certain round slot if the frame has to be taken out of a round snap shot.

➤ Name `slot`
Type `Link`
Description `Slot` linked to `R_slot`.

`TTA.Reinit_Reintegration`: output attributes

Scope: TTP˙Plan framework scope

➤ Name `ox`
Type `Int`
Description Object index. For each object type, `ox` is a number unique for each instance between `0` and `<Type.count> - 1`.

`TTA.Sched_Step_CS`: output attributes

Scope: TTP˙Plan scope

➤ Name `ox`
Type `Int`
Description Object index. For each object type, `ox` is a number unique for each instance between `0` and `<Type.count> - 1`.

➤ Name `step_nr`
Type `Int`
Description The schedule step this checksum belongs to.

`TTA.Slot`: output attributes

Scope: TTP˙Plan scope

➤ Name `duration`
Type `MT_Time_Span`
Description Duration of slot in macroticks (difference between the action times of this slot and of its successor).

➤ Name `idle_time`
Type `MT_Time_Span`
Description Length of pause (outside transmission time and inter-frame gap) between this and the next slot (in macroticks).

➤ Name `max_frame_size`
Type `Bit_Size`
Description Size of the biggest N-frame transmitted in this slot, calculated by the scheduler (in bytes:bits).

➤ Name `net_duration`
Type `MT_Time_Span`
Description Length of transmission phase of slot plus inter-frame gap (in macroticks).

➤ Name `number`
Type `Int`
Description Index of slot in round.

➤ Name `ox`
Type `Int`
Description Object index. For each object type, `ox` is a number unique for each instance between `0` and `<Type.count> - 1`.

➤ Name `r_phase`
Type `MT_Time_Span`
Description Phase of slot within the round (in macroticks).

➤ Name `r_slots`
Type `Link_Set`
Description `R_slots` linked to `Slot`.

➤ Name `transmission_time`
Type `MT_Time_Span`
Description Time needed for the transmission of the largest frame transmitted in this slot (in macroticks).

`TTA.Subsystem`: output attributes

Scope: TTP Plan scope

➤ Name `ox`
Type `Int`
Description Object index. For each object type, `ox` is a number unique for each instance between `0` and `<Type.count> - 1`.

`TTA.Subsystem_Status_Message`: output attributes

Scope: TTP Plan scope

➤ Name `ox`
Type `Int`
Description Object index. For each object type, `ox` is a number unique for each instance between `0` and `<Type.count> - 1`.

`TTA.Virtual_Controller`: output attributes

Scope: TTP Plan framework scope

➤ Name `ox`
Type `Int`
Description Object index. For each object type, `ox` is a number unique for each instance between `0` and `<Type.count> - 1`.

`TTA.Web`: output attributes

Scope: TTP Plan scope

➤ Name `max_delay`
Type `Nano_Time_Span`
Description Maximum delay for all hosts and channels (in nanoseconds).

➤ Name `min_delay`
Type `Nano_Time_Span`
Description Minimum delay for all hosts and channels (in nanoseconds).

➤ Name `ox`
Type `Int`
Description Object index. For each object type, `ox` is a number unique for each instance between `0` and `<Type.count> - 1`.

`TTA.X_Frame`: output attributes

Scope: TTP Plan scope

➤ Name `c_state_size`
Type `Bit_Size`
Description Length of C-state data carried by frame (in bytes:bits).

➤ Name `channel`
Type `Channel`
Description Channel on which the frame is transmitted.

➤ Name `cni_address`
Type `Hex`
Description Offset of this frame to the CNI message area base address.

➤ Name `data_offset`
Type `Int`
Description Offset of Remote Pin Voting and message data in frame (in bits).

➤ Name `data_size`
Type `Bit_Size`
Description Length of message data (in bytes:bits). This includes any packing holes.

➤ Name `host`
Type `Link`
Description `Host` linked to `Frame`.

➤ Name `mode_number`
Type `Int`
Description `Cluster_Mode` the frame belongs to.

➤ Name `ox`
Type `Int`
Description Object index. For each object type, `ox` is a number unique for each instance between `0` and `<Type.count> - 1`.

➤ Name `r_slot`
Type `Link`
Description `R_slot` linked to `Frame`.

➤ Name `round`
Type `Int`
Description Number of round the frame is transmitted in.

➤ Name `rpv_size`
Type `Bit_Size`
Description Length of Remote Pin Voting data (in bytes:bits) carried by frame.

➤ Name `status_offset`
Type `Int`
Description Offset of status data in frame (in bytes).

`TTP.Clock_Sync_Multiweb`: output attributes

Scope: TTP˙Plan framework scope

➤ Name `ox`
Type `Int`
Description Object index. For each object type, `ox` is a number unique for each instance between `0` and `<Type.count> - 1`.

`TTP.Clock_Sync_Standard`: output attributes

Scope: TTP˙Plan framework scope

➤ Name `ox`
Type `Int`
Description Object index. For each object type, `ox` is a number unique for each instance between `0` and `<Type.count> - 1`.

`TTP.Controller_T3C2NF`: output attributes

Scope: TTP˙Plan framework scope

➤ Name `ox`
Type `Int`
Description Object index. For each object type, `ox` is a number unique for each instance between `0` and `<Type.count> - 1`.

➤ Name `user_interrupt_1`
Type `User_Int_Spec`
Description Specification for User Interrupt 1 in `Working_Mode`.

➤ Name `user_interrupt_2`
Type `User_Int_Spec`
Description Specification for User Interrupt 2 in `Working_Mode`.

`TTP.Firmware_C2NF_2_0`: output attributes

Scope: TTP Plan framework scope

➤ Name `ox`
Type `Int`
Description Object index. For each object type, `ox` is a number unique for each instance between `0` and `<Type.count> - 1`.

`TTP.MHub`: output attributes

Scope: TTP Plan scope

➤ Name `offset`
Type `Micro_Time_Span`
Description Offset (in us) of the SOF between Master Node and M-Hub. This marks the first switching instant in the schedule table and allows the M-Hub to determine its current temporal position in the schedule.

➤ Name `ox`
Type `Int`
Description Object index. For each object type, `ox` is a number unique for each instance between `0` and `<Type.count> - 1`.

➤ Name `tick_ns`
Type `Int`
Description Length of a tick (in ns).

`TTP.Monitor_Host_TTPpowernode_C2NF`: output attributes

Scope: TTP Plan framework scope

➤ Name `frames`
Type `Link_Set`
Description `Frames` linked to `Host`.

- Name `ignore_lifesign`
  Type `Boolean`
  Description If set, the controller will always send frames with the 'host-alive'-bit set, independent of whether the host has updated the lifesign or not.
- Name `last_schedule_step`
  Type `Int`
  Description Indicates the latest schedule step this host participated in.
- Name `membershipbit`
  Type `Int`
  Description Bit number of this host in the membership vector.
- Name `ox`
  Type `Int`
  Description Object index. For each object type, `ox` is a number unique for each instance between `0` and `<Type.count> - 1`.
- Name `permutation`
  Type `Code_Generator_Permutation`
  Description Specifies which permutation the code generator should consider. Currently the values are * 0,1 - no permuation * 1,0 - permutation * hardware - use specification in node config
- Name `schedule_cs`
  Type `Long-Int`
  Description Checksum of last (unfrozen) schedule made.
- Name `startup_timeout_MT`
  Type `MT_Time_Span`
  Description Unique amount of time (in macroticks) the controller waits until it sends the first cold-start frame.

`TTP.Msg_Area_Page`: output attributes

Scope: TTP˙Plan framework scope

- Name `begin`
  Type `Hex`
  Description Start address of CNI-page.
- Name `end`
  Type `Hex`
  Description Last address of CNI-page.
- Name `ox`
  Type `Int`
  Description Object index. For each object type, `ox` is a number unique for each instance between `0` and `<Type.count> - 1`.

`TTP.Protocol_V_0_6`: output attributes

Scope: TTP Plan framework scope

➤ Name `ox`
Type `Int`
Description Object index. For each object type, `ox` is a number unique for each instance between `0` and `<Type.count> - 1`.

13. Using the TTP M-Hub in TTP-Plan

This section describes the usage and settings for the TTP M-Hub in TTP-Plan. For a detailed description of the TTP M-Hub refer to the *TTP Hub/Codec Interface Control Document (ICD), D-115-G-10-016*.

The TTP M-Hub acts as a star coupler connecting selected hosts of the cluster. It has seven ports, of which port 0 is reserved for the so called *Master Node*, which is directly attached to the TTP M-Hub. The other six ports can be used by all other hosts. Several hosts can be connected to one port if they form a *branch*, but as usual in TTP, only one host at a time is allowed to transmit on one port in one sending slot.

Prior to defining the TTP M-Hub in TTP-Plan after hardware installation, the `TTP_MHub` plugin for TTP-Plan must be installed and loaded. Then proceed as follows:

1. Define your TTP cluster as usual.
2. Create a `TTP.MHub` object using the 'Edit.Objects' menu.
3. Connect the desired hosts to the TTP M-Hub by defining `TTP.Host_on_MHub` links via 'Edit.Links'. Specify the port each of them is connected to. Note that the designated Master Node has to be connected to branch/port `0`.
4. Make sure the connected hosts are in the same web as the TTP M-Hub (`TTA.Host_in_Web`) and that the `transmission_speed` of the web and the TTP M-Hub is the same (`TTA.Web` and `TTP.MHub`).
5. Open the selector of `TTA.Slot` and select the slot associated to your Master Node. Set the attribute `sort_key` to a very low value, e.g., `-99`, to ensure that the Master Node is the first one to send, and thus gets to send in slot `0` as is required by the TTP M-Hub design..
6. Create a cluster schedule and save your CDB.
7. Create the configuration for the TTP M-Hub using 'Schedule.Generate M-Hub configuration'. The configuration – containing both global parameters and the schedule table, as defined in the ICD – is exported as a header file containing a C array. It can then be compiled by the application and downloaded to the TTP M-Hub.

   Note: The configuration file is not stored in the CDB, even if you save your cluster design *after* you have created the configuration. Only the M-Hub-specific object-model parts are stored.

A. Controllers Supported by TTP-Plan

Different communication controllers may have different application areas and capabilities, the most prominent of them being the IFG (inter-frame gap) duration. This is the duration needed for the computation of the frame status and the update of the CNI between any two transmissions. The minimum slot duration of a cluster is limited by the slowest controller in the cluster, i.e., the longest required IFG. Another significant property that may differ between controller implementations is the supported transmission speed (or bit rate) on the bus — some may be able to transmit data at 25 Mbit/s.

Since the choice of controllers used for a cluster obviously has an influence on the scheduling, the user must tell TTP-Plan about the controller type to be used; this property is set in the `controller_type` attribute of the `Cluster` object.

Three different strategies can be employed here:

- A 'real' controller is used as basis for scheduling. This is the normal way if a schedule is designed for a specific hardware. The generated schedule will be executable by this controller implementation.
- A 'controller set' is used for clusters with mixed controller types. The compatible properties are calculated automatically. Edit the settings for a mixed cluster as follows:

  Step 1: Set the attribute `controller_type` to `Controller_Set` (default value is `TTTech_C2NF`).

  Step 2: Click 'Edit.Link' and select

  `TTA.Controller_Set_uses_Controller_R`

  from the menu.

  Step 3: Choose `Controller_Set` in the `controller_set` field and the controller types that you use in the `element` field, then create links between them by clicking 'New' for each link.
- Alternatively, the 'virtual controller' can be used (by setting `controller_type` to `Virtual_Controller`). For this controller *any* settings for supported bit rates on the bus, duration of the IFG, etc., can be selected. The resulting schedule may not be executable by physically existing controller implementations, and no MEDL can be generated for the virtual controller.

The ‘virtual controller’ allows to create schedules for controller implementations that are not yet available on the market, but for which the basic properties (IFG duration, supported transmission speeds) are already known. With this information it can thus be tested whether the bandwidth requirements of a specific application will be met by such a controller.

Currently, the following ‘real’ controllers are supported by TTP-Plan:

A.1. `TTTech_C2NF`

Official name: AS8202NF controller.

- Dedicated communication controller supporting TTP communication protocol.
- Suited for dependable distributed real-time systems with guaranteed response time.
- Application fields: automotive (by-wire braking, steering, vehicle dynamics control, drive train control), aerospace (aircraft electronic systems), industrial systems, railway systems.
- Asynchronous data rate up to 5 Mbit/s (MFM / Manchester).
- Synchronous data rate 25 Mbit/s (MII) for firmware 2.*.
- Supports multiplexing, X-frames, Remote Pin Voting.
- CNI consists of 7 individually write-protectable 4kByte-pages.
- Automotive qualified; temperature range: -40C to 125C.

B. The “Goodies” Scripts of TTP-Plan

The “Goodies” directory of TTP-Plan contains a variety of helpful Python scripts.[64] By default, it is located in

`C:\TTTech\TTP\<toolname>\<version>`

and can be accessed via the ‘Scripts’ menu of TTP-Plan. Currently it contains the following scripts:

- `adjust_message_period.py`
  For all message transmissions in the cluster, the value of ‘`d_period`’ has to be at least as large as the ‘`tr_period`’ of the cluster. If it is smaller, this script changes it to the duration of the `tr_period`.
- `change_init_values.py`
  This script changes the `init_value` of all messages (even those not yet having one defined) to the specified value.
- `clear_view_instrument.py`
  For each message type it has to be defined how it should be presented in TTP-View (i.e., a HEX number, an integer, etc.). This script resets the presentation to the default TTP-View instrument.
- `copy_message.py`
  Creates an exact copy of an existing message (i.e., same data type, same subsystem, same cluster mode etc.), but with a new name. This speeds up the process of creating several similar messages because you do not need to start from scratch each time. You just copy an existing message and then change the necessary parameters.
- `count_objects.py`
  Counts all objects that have been defined, and shows how many there are in the database.
- `create_disturbance_node.py`
  Creates a disturbance node.
- `deepcopy_subsys.py`
  Copies a subsystem with all its properties and messages.

[64] See Section 4 for details about Python and scripts.

- `equalize_slot_lengths.py`
  Sets the slots for all nodes to the same length by dividing the duration of the round by the number of slots.
- `export_host.py`
  Writes a Python script containing the definition of a specific host and all its subsystems, messages, and message types.
- `Hosts.py`
  Writes the names of the hosts and the numbers of their sending slots to a file named ‘`hosts.txt`’.
- `normalize_message_period.py`
  Sets the `d_period` to an integer multiple of the `tr_period`. This makes the job of the scheduler easier.
- `set_all_messages_for_nonredundant_transmission.py`
  Sets the redundancy degree of all messages to ‘1’, i.e., transmission over only one channel.
- `set_all_messages_for_redundant_transmission.py`
  Sets the redundancy degree of all messages to ‘2’, i.e., transmission over both channels.
- `set_delay_matrix.py`
  Sets the delay matrix (i.e., the transmission delays (in ns) between all nodes in both directions) for a web. By default, all values in the matrix are zero, different values have to be specified by the user in a CSV file. Execute “Make new schedule” after loading the delay matrix for the values to take effect.
- `set_init_values.py`
  For all messages that do not yet have the attribute defined, this script sets the `init_value` to a defined value. See also “`TTA.Message`” in Section 11.1.
- `set_min_slot_duration_to_c_state.py`
  Sets a minimum slot width. Normally, the slot width is a function of the amount of data to be transmitted in it. However, during Startup Mode even I-frames have to be transmitted, hence some space must be reserved for them as well. Otherwise the affected node can only be passive during startup and has to integrate later.
- `set_serial_number.py`
  Numbers all nodes consecutively. This saves you the trouble of setting the required `serial_no` attribute.
- `use_x_frames.py`
  Earlier, there were some controllers that could not handle X-frames. Now that the newer ones can, this function has to be activated for them by means of this script.
- `write_msg.py`
  Generates a file ‘`msg.info`’ containing a list of all messages with originating subsystem, length and period.

Those of the goodie scripts that require a loaded database before they can be executed are guarded by a checking function named `model_exists`. In case you try to run such a script without loading a database first, TTP-Plan will recognize this and remind you to do so.

C. Glossary and Abbreviations

APDB (Application Descriptor Block) Memory block containing information about the application (for the Bootloader); for example, hook addresses and the cluster-wide identification number of each node.

API (Application Programming Interface) A set of functions, constants and data types that provide the interface to a software module.

Application In the context of the TTP-Tools this denotes the user-programmed application code that runs on the cluster.

Application cycle A node-local task scheduling unit; the cycle where a sequence of tasks on a host is repeated; for simplicity, this is an integer number of *cluster cycles*.

Application task An application task is a user-defined procedure that is activated by the operating system in a time-triggered way. See also **Task**.

Array Structured data type consisting of several elements of the same type (for example, a number of integer values).

Association Functional connection between classes of the object model, for example, `TTA.Task_uses_Message`. An association comprises a number of instances called *links*, which connect the individual *objects*.

Atomic data types Also called "primitive". These data types are read by the CPU in one single access (in contrast to *structured* data types, such as arrays).

Attribute Parameter of an object or link. Some can be edited by the user (required or optional), whereas some are calculated by the tool (for example, during scheduling).

Batch mode The batch mode of the tool is invoked from the command line and allows the execution of commands or scripts without intermediate user interaction.

Bus A communication system topology in which nodes are directly connected to a single, common communication media (as opposed to connection through stars, gateways, etc.). The term "bus" is also used to refer to the media itself, in which case it denotes the physical connection between the nodes of a cluster, over which the data communication takes place. The latter is the usual meaning in the context of the TTP-Tools.

Byte order Also *endianness*. The property of a hardware or software architecture that indicates how multibyte values are represented and stored (for example, big endian and little endian).

C-state (Controller state) The internal state of the communication controller. It describes the view of a single controller over the cluster. These local views need an agreement (the acknowledgment and clique-detection mechanism) to be consistent in the cluster. This should guarantee that only controllers with the same view (i.e., C-state) are participating in the cluster. An agreed C-state may be seen as "cluster state".

The C-state comprises information on the current global time, the current position in the MEDL, the current operating mode, and the current membership.

CDB (Cluster Database) Database containing the cluster design.

Channel Also *communication channel*; a bus typically consists of two channels for redundant data transmission, hence two frames can be sent in each round-slot; one per channel.

Class Definition of a certain type of *objects*. The actually created objects (*instances*) are based on the class, but contain additional (individual) information. Their common properties are defined in the class itself.

Cluster A cluster comprises a set of nodes (SRUs or ECUs) that share a bus in a communication system. A cluster describes the entire system.

Cluster cycle The sequence of different rounds in a cluster.

Cluster schedule A "timetable" describing the temporal sequence of the messages transmitted within the cluster. Automatically generated by TTP-Plan after the cluster design has been completed.

CNI (Communication Network Interface) The interface between the host computer and the communication controller within a node of a distributed system. Working as a buffer between the bus and the application, the CNI exchanges data between the two systems. The CNI is primarily used by the FT-COM layer to receive and send data.

Cold start Communication startup, the time from "power on" to first available use of cluster. Usually initiated by a so called *cold-start node* sending startup frames.

Communication controller See **Controller**.

Connector An electrical connector is an electro-mechanical device for joining electrical circuits as an interface using a mechanical assembly. In computing, an electrical connector can also be known as a physical interface (see also physical layer).

Controller The part of a node that controls the communication within the cluster. It is a piece of hardware that contains the specification of when each message must be sent. Also called *communication controller*.

CRC (Cyclic Redundancy Code) A checksum transmitted together with each frame. It allows the frame to be checked for correctness at the receiver by means of the *cyclic redundancy check* (sometimes – falsely! – *also* called CRC).

Deadline The absolute time within the application cycle when the task must have finished execution.

Design period (`d_period`) The maximum transmission period of a message. During scheduling, it is used as input for the calculation of the *actual* transmission period (`a_period`).

Download The process of transferring data to a node's non-volatile memory via the bus.

ECU (Electronic Control Unit) A node (computer).

Editor A window in the graphical user interface of the tool. It allows the editing of object or link attributes, or even schedules.

Endianness See **Byte order**.

Essential model The classes and associations shown in the pilot, representing the parts of the object model that can be accessed by the user.

Filter A function (button) in the editors of the tool, which allows you to restrict the list of displayed items according to different selection criteria (for example, all messages sent by a certain host).

Frame A frame is a complete unit of transmission which comprises all information transmitted with an identifier in one slot, on one channel. Frames usually comprise a header, the application data (messages), and a cyclic redundancy code (CRC) to check the correct transmission of data on the receiver side.

Besides the application data, the frame contains data required internally within the cluster to ensure proper operation of the communication protocol. The bit stream on the communication bus consists of consecutively sent frames, separated by inter-frame gaps (IFG). There are different types of frames with specific properties for different purposes.

Frame buffer A frame buffer is a RAM copy of a frame. The FT-COM layer works on these copies, writing and reading messages, because the access to the RAM is much faster than that to the CNI.

FT-COM (Fault-Tolerant Communication) layer The fault-tolerant communication layer performs message and frame handling and message level redundancy management. TTTech's FT-COM implementation provides numerous functions, such as startup and reintegration support, byte order swapping, lifesign update, packing and unpacking of messages, channel redundancy management, and calculation of replica-deterministic agreements.

The FT-COM layer isolates the application from the time-triggered communication network; the application only accesses messages, not the communication controller.

FT-task Special task, generated by TTP-Build, that performs operations of the FT-COM layer.

Gateway A node that participates in two or more clusters and thus allows the exchange of information between clusters.

Ghost Short for `MUX_Ghost`; a "virtual" host used to model complex behavior in the context of multiplexing.

Global time See **Reference time**.

Goodies A collection of Python scripts that facilitate or speed up several procedures for the user. They can be accessed via the 'Scripts' menu or from the command line.

GUI (Graphical User Interface) Interactive interface of the tool, allowing for direct manipulation via menus and buttons.

Guide An interactive tutorial in the form of a step-by-step guide that leads the user through the basic steps of a cluster and/or node design in order to get familiar with the tool.

H-state (History state) The data elements that a task updates during execution, and that influence the next iteration of execution. (Example: an integration algorithm keeps the "I" element in an h-state data element, i.e., a static variable/buffer.) During startup, the h-state is initialized with **init values**; from then on, it is periodically updated by the periodic execution of the algorithm.

H-state data elements are required for proper operation of the algorithm ("static memory" of a task) and need to be re-initialized or recovered in case of a temporary crash of the function. The FT-COM layer supports management of h-state data by storing it not in the task itself, but in the FT-COM layer; in this way, reintegration of replica-deterministic subsystems with h-state is supported.

Hook A hook is a function which is called if a well-defined system condition occurs. Inside a hook, a user-defined function can be called or user-defined functionality can be executed. Generally, a hook can be used for a customized reaction to a system condition.

Host The computational unit within a node that executes the operating system and application software.

Host lifesign field A field in the CNI that must be periodically updated by the host (at least once between two transmissions of its controller) to show host activity (see **lifesign**).

HS-COM (High-Speed Communication) layer The HS-COM layer is an IP module which is optimized for high-speed message processing. It provides asynchronous access to the TTP data and is capable of handling 32-bit and 64-bit state messages, as well as 128-bit event messages. The HS-COM plugin of TTP-Build consists of an interrupt scheduler and a code generator. The configuration for the HS-COM layer is calculated by TTP-Build based on the node design.

HW-COM (Hardware Communication) layer The HW-COM layer is an IP module which is optimized for fast and memory-efficient message handling (32-bit data blocks only) and provides asynchronous access to the TTP data. The HW-COM plugin of TTP-Build consists of an interrupt scheduler and a code generator. The configuration for the HW-COM layer is calculated by TTP-Build based on the node design.

I-frame (Initialization frame) This type of frame contains the entire C-state, but no data. I-frames are needed to start up the system and to integrate nodes which are not yet actively participating in transmission, or to reintegrate nodes which are no longer actively participating.

IFG (Inter-Frame Gap) Time between two consecutive frames on the bus.

Initialization value (`init_value`) A "safe" value that a message can start with when no current data is (yet) available, for example during startup or reintegration.

Instance Specific representative of a class or an association. For example, the messages m1, m2, m3 are all instances of the class `TTA.Message`. An instance of a class is called *object*, an instance of an association is called *link*.

Interrupt A hardware-supported request for a specific task activation, caused by an event outside the currently active computation.

Invariant A logical condition which must be satisfied at all times. A violation of such a condition results in an error.

IVN Interface Version Number.

Lifesign A periodic signal indicating activity.

Link An instance of an association.

Local time A time source which is provided by the host CPU. The local time of a node, which is always available, is relevant for asynchronous operations, for example, when – during node startup or during a temporary dropout of the global (reference) clock – some tasks must be executed periodically, but without global synchronization.

Macrotick (MT) A unit of global time used to specify the current time and a desired progression of time. Each macrotick is made up of a number of *microticks*.

MAN Manchester encoding.

MEDL (Message Descriptor List) The MEDL describes the communication pattern of a cluster by defining the round layout and the parameters required for clock synchronization. The contents of the MEDL determine when a frame has to be sent or received.

Membership The information on which nodes in the cluster are currently operational, based on the correct transmission activity.

Message An element of data communication. It is defined by a data type (*message type* and (optionally) an init value. In a time-triggered context, messages have *state* semantics (*state message*), which means each message contains information about the current state of a system parameter (like 'current temperature' or 'current pressure'). They are strictly periodic (i.e., they have an update period), non-blocking, and have only one sender/producer (but potentially many receivers/consumers). They also have optional schedule-relevant attributes, such as the *validity span*.

In TTA systems, you can further distinguish between the following types of messages:

- *global*: transmitted on the bus, their size on the bus needs to be defined (attribute `type_length`)
- *local*: node-internal messages between tasks, not transmitted on the bus
- *I/O*: necessary to define access to sensors and actuators

The application in a TTA system does not differentiate between global and local messages, which makes it easy to move tasks without changing the application.

Message age Indicates how many rounds ago in the cluster cycle the latest valid copy of this message was received. The age of a message is counted in inverse order to the *message generation*, i.e., the 'youngest' message generation has the age '0'.

Message buffer A memory area where copies of message instances are placed for further processing (for example, for calculation of an RDA).

Message generation Messages are sent periodically. Each time a message is sent, this is called a *message generation*. The oldest message of a cluster cycle thus is of generation '0'.

Message period Time between two consecutive transmissions of the same message.

Message status The status of a message in a TTA network; for global messages, it consists of the (optional) *sender status* and the *receiver status* computed from the network's frame status information.

Message transmission interval Time interval between the *latest* update of a message in the *earliest* slot of the round and the *earliest* update in the *latest* slot. This is the time where the current message generation is available on the bus. The transmission interval is required to determine the earliest start and latest end of the application tasks.

Message type The data type of a message. There are *atomic* (or primitive) and *structured* (complex) data types, see also Section 11.1.

Microtick (uT) A microtick is a periodic signal that is generated by the oscillator of the communication controller. It is a basic unit of time measurement, and its length can be different on different nodes. Each macrotick is made up of a number of controller microticks.

Monitor MEDL MEDL of the Monitoring Node, generated automatically by TTP-Plan after scheduling.

Multiplexing Multiplexing means that two or more hosts use the same sending slot in different rounds. With multiplexing, the number of physical nodes in a cluster can exceed the number of slots in a round.

N-frame (Normal frame) N-frames are used to transmit data between the nodes. They consist of frame header, (optional) network management vector, application/payload data (messages), and a cyclic redundancy code (CRC).

NDB (Node Database) Database containing the node design (task schedule, FT-COM code etc.).

Node (1) A node is a computer which comprises one "embedded" CPU type (*host*) and the communication controller. A node is also referred to as *smallest replaceable unit (SRU)* or *electrical control unit (ECU)*. A cluster consists of several nodes, communicating with each other via the bus.

Node (2) In the treeview of a browser, the "forks" where the different branches begin are also called *nodes*. This term is not to be confused with the nodes (computers) of a cluster (see Node (1))!

Object An instance of a class.

Period Also *Message period*, the time between two consecutive transmissions of the same message.

Phase Generally, an offset within a time period. Depending on the context, it can mean different things: the phase of a *message* defines where in the cluster cycle a message is sent (relative to the round). The phase of a *task* defines the offset of the task start relative to the application cycle.

Physical interface Synonym for physical layer.

Pilot Interactive display of the essential object model as interconnected rectangles. Allows for direct graphical access to the editors of all classes and associations.

Python Interpreted, object-oriented programming language. Provides the possibility to write your own scripts for execution in the tool (for example, startup scripts).

Raw value The message instance (value of a replica) as it is received, i.e., without any agreement being calculated.

RDA (Replica Deterministic Agreement) An RDA takes the raw message values of all active replicas and computes an agreed result. This procedure is necessary to ensure replica determinism.

Receiver status Counter that is initialized to zero at the beginning of the message transmission interval (see there), and increased by one for each message copy that is received correctly. It tells the application about the availability of the message.

Reference time Reference time refers to a time source provided by the controller. It is synonymous with the term *"global time"*. The reference time is used for tasks which only should be executed when TTP is operating (for example, FT-tasks).

Reintegration When a node fails temporarily and restarts, its controller starts to listen to the bus. Depending on the ongoing communication, it either performs a *cold start*, or it gets hold of the *C-state* and *reintegrates*. After this, its subsystems also need to reintegrate on the "software level" to become active members of the cluster again. For this, there are different strategies (not to reintegrate, to reintegrate on **h-state**, or to reintegrate with an **init value**).

Replica A redundant subsystem on one node.

Replica determinism A desired relation between replicas. A set of replicas is *replica-deterministic* if all of them have the same visible external *h-state* and produce the same output messages at points in time that are at most an interval of d time units apart.

Replica round Denotes the round where each node that contributes to the RDA must send exactly once (even if the actually used agreement would not require all of them). Not necessarily equidistant.

Round Its duration defines the shortest possible re-transmit period of any message in the cluster. It is specified in the cluster attribute `tr_period` which defines the minimum of all `a_periods` in the cluster.

Round-slot Sending slot of a certain node within a round. Each round-slot may contain two frames, one for each channel.

RPV (Remote Pin Voting) A voting mechanism that enables a group of nodes within the cluster to bring a faulty node into a safe state without that node's cooperation.

Schedule In the context of TTP-Plan, this denotes the *cluster schedule*.

Schedule extension Scheduling is done in discrete steps. Each time you make a schedule, this is called a "schedule step". You can keep a schedule (by "freezing" the current schedule step) and add new messages to it, thus filling the "holes" in the original schedule without changing it.

Schedule table A set of structures (tasks and task chains), used by the operating system for task activation.

Script A (Python) file containing a sequence of commands, used to execute repetitive actions automatically. With *batch scripts* such sequences can be executed without intermediate user interaction.

Sender status Specifies if a sender-status bit is transmitted along with the message. The sender-status bit allows the sending host to tell the receivers of the message whether the message value is valid or not.

Slot A slot is the smallest time interval of a TDMA schedule. In each slot, one statically defined node is allowed to access the communication medium and transmit frames. The sequence of slots within a cluster is called a *round*.

Sort key In the link selector, the 'sort key' is the object column you want to sort the links by.

SRU (Smallest Replaceable Unit) A node; an electronic module that is connected to a cluster.

Startup, asynchronous This method enables an application to be started before the global/reference time source is established. The application will run on the local time until the global time is started. Then the local time is synchronized with the global time.

Startup, synchronous During synchronous startup, the cluster starts immediately on its local time, but the application is first started (by the operating system) when a global time has been established (if the global time has not been established, no application functions can be run). Then the local time source of each node is synchronized with the global time source.

State message A state message contains "state information", e.g., about the current state of a system parameter ("current temperature"). This information is only valid for a certain time, i.e., it has a *validity span*. An "event message", on the contrary, is sent according to the occurrence of an event and has no validity span.

Struct Structured data type; a logical unit of a fixed number of elements of potentially different types, e.g., a set of data describing a certain state (time, temperature, speed, on/off, etc.).

Structured data types Data types consisting of several elements. In case of *arrays*, they contain elements of the same type, in case of *structs*, elements of different types.

Subsystem A set of tasks that together provide a specific functionality within the distributed system. Several subsystems may be executed on one host, but one subsystem may also be executed simultaneously on several hosts. In TTP-Plan a subsystem is defined by a name and the messages it sends into the system.

Subsystem status The application can set a "status" for a subsystem; this is necessary for h-state reintegration. In this case the FT-COM layer takes care that the sub-

system gets its h-state. Only then is the first task of the subsystem started and the subsystem status set to 'valid'. This means that this particular subsystem does not have to wait for h-state information from other nodes (where this subsystem runs and the subsystem status is valid) to reintegrate.

Task A time-invariant description of a specific function. A task is a functional unit and is executed independently of other tasks. In a system, the tasks are mostly grouped into subsystems. Each task is part of exactly one subsystem, but each subsystem may contain as many tasks as necessary. There are *application tasks* and *FT-tasks*.

TD-COM (Table-Driven Communication) layer The TD-COM plugin consists of an interrupt scheduler and a code generator. The configuration code for the TD-COM layer is calculated by TTP-Build based on the node design.

TDMA (Time Division Multiple Access) Collision-free communication scheme where the time axis is partitioned into slots. Each node in a cluster may only send messages during its exclusive sending slot.

Tick See **Macrotick**.

Time budget The amount of CPU time a task is allowed to utilize for one execution. Note that this refers to actual CPU usage and may differ a lot if a task is executed on different hardware.

Transceiver In TTP context, this mostly denotes an *SFPtransceiver*, which is a compact, hot-pluggable transceiver used for both telecommunication and data communications applications. It interfaces a network device mother board (for a switch, router, media converter or similar device) to a fiber optic or copper networking cable. The SFP transceiver contains a PCB (printed circuit board) that mates with the SFP electrical connector in the host system.

TTA (Time-Triggered Architecture) Architecture for distributed real-time systems in safety-critical applications.

TTP (Time-Triggered Protocol) Communication protocol where the point in time of message transmission is derived from the progression of the global time (reference time) of the system.

Validity interval Time during which a *message generation* is valid.

Validity span Period of validity of a message instance (by default the time between transmission and retransmission, but it can also be shorter or longer). The message may only be used during this time.

WCET (Worst Case Execution Time) The maximum amount of CPU time that will be utilized for one execution of a certain task.

WDB (WCET Database) Database containing the measured execution times of functions of the FT-COM layer and the operating system. Generated by TTP-Profile.

Web A set of hosts within a cluster, connected by an independent physical medium, e.g., one bus. Its attributes are mainly designed for specification of different aspects of data transmission and the physical interfaces. All the webs present in a cluster are assumed to be synchronized (precision PI) to each other.

X-frame (Extended frame) X-frames contain initialization information (I-frames) and application data (N-frames), with each part being secured with a CRC. Clusters which utilize this type of frame have a higher transmission overhead due to the inclusion of the C-state in each frame, but allow for schedules where all nodes can transmit data in every slot in every round.

Index

— B —

— C —

— F —

— G —

— N —

— O —

— P —

— S —

— T —

— U —

— V —

— W —

— X —

Disclaimer

While every precaution has been taken in the preparation of this document, the publishers assume no responsibility for any remaining errors or omissions, or for damages resulting from the use of the information herein.

 Document number D-001-G-01-003