SIEMENS

SIMATIC IT Unilab 6.7

Concepts Guide - part 3

Concepts Guide

Edition 09/2014

A5E00428244-09

Guidelines

This manual contains notices intended to protect the products and connected equipment against damage. These notices are graded according to severity by the following texts:

Caution

Indicates that if the proper precautions are not taken, this can result into property damage.

Notice

Draws your attention to particularly important information on handling the product, the product itself or to a particular part of the documentation.

Trademarks

Disclaimer of Liability

Siemens AG
Automation and Drives
Postfach 4848
90437 NÜRNBERG
GERMANY

A5E00428244-09
09/2014

Copyright © Siemens AG 2013
Technical data subject to change

Preface

Where is this manual valid?

This manual is valid for release 6.7 of SIMATIC IT Unilab.

Basic knowledge required

This guide is intended for SIMATIC IT Unilab users who are responsible for system configuration, such as application managers and system integrators (consultants).

To be able to understand the concepts and examples discussed in this guide, the reader should at least have taken the SIMATIC IT Unilab Basic Training.

Purpose

This Concepts Guide explains all concepts of SIMATIC IT Unilab and includes several scenarios that illustrate how these concepts can be used in specific situations.

SIMATIC IT Documentation Library

The SIMATIC IT Unilab Documentation Library provides you with a comprehensive and user-friendly interface to access the overall product documentation where manuals and helps online can be browsed by functionality or by component.

Readme

The installation includes a readme file, which contains information on upgrade procedures and compatibility with previous releases. This file is supplied both in standard text (**Readme.wri**) and in Acrobat PDF (**Readme.pdf**) format.

This file is available in folder \ReleaseNotes of the setup DVD and is available from the SIMATIC IT Unilab Documentation Library.

Related documentation

The **Unilab Concepts Guide parts 1** and **2** contain information related to the content of this Concepts Guide.

All these documents are available online in the Unilab Product Library.

Conventions

The table below describes the specific typographic conventions that are used throughout this manual:

Symbol/Convention	Indicates...
E.g.	Where examples are given.
Text in bold	The names of menus, commands, dialog boxes and toolbar buttons and, in general, all strings (e.g. **File** menu; **Save** command).
KEY1+KEY2	Shortcut keys, which permit rapid access to commands (e.g. CTRL+C).
UPPERCASE	The names of keyboard keys (e.g. RETURN key).
Italics	Noun with special importance or significance for which emphasis is needed. The names of parameters that must be replaced with a specific name or value.
>	A succession of commands in which the command preceding the symbol must be selected before the command following it.
`Code example`	Code example.

Acronyms and abbreviations

The table below lists the acronyms and abbreviations that are used throughout this manual:

Acronyms / Abbreviation	Meaning
ad	Address
API	Application Program Interface
ar	Access rights
au	Attribute
ca	Intervention
cf	Custom function
cn	Condition
cs	Condition set
db	Database
dd	Data domain
DLL	Dynamic Link Library
eq	Equipment
ev	Event
fa	Functional access
FIFO	First In First Out
freq	Frequency
gk	Group key
GUI	Graphical User Interface
hs	History
ic	Info card
id	Identification

Acronyms / Abbreviation	**Meaning**
ie	Info field (Configuration)
ii	Info field (Operational)
ip	Info profile
lc	Life cycle (or life cycle model)
LIMS	Laboratory Information Management System
ly	Layout
lo	Storage location
me	Method (Operational)
MES	Manufacturing Execution System
MRP	Materials Requirement Planning
mt	Method (Configuration)
pa	Parameter (Operational)
pg	Parameter group
pp	Parameter profile
pr	Parameter Definition
pref	Preference
pt	protocol
rd	Raw data
RDBMS	Relational Database Management System
rq	Request
rt	Request type
sc	Sample code
sd	study
seq	Sequence (number)
SOP	Standard Operating Procedure
spec	Specification
ss	Status
st	Sample type
tk	Task
tp	Time point
uc	Unique code mask
up	User profile
us	User
ws	Worksheet
Wt	Worksheet type

SIMATIC IT Training Center

Siemens IA AS MES offers a number of training courses to familiarize you with the SIMATIC IT product suite. To successfully achieve this goal, training consists of lessons in both theory and practice.

Courses are held year-round, according to a program that is published well in advance of the first scheduled session.

The material on the basis of which our courses are conducted reflects the result of years of experience in process, LIMS, quality control and production management.

All courses are held by expert personnel that are aware of the developments and innovations in the Siemens IA AS MES product suite.

Courses are held in English at the Siemens IA AS MES Training Centers.

Upon request, training courses can also be organized on the customer's premises.

For more information on the training course calendar, please visit our technical web site (www.siemens.com/simatic-it/training).

SIMATIC IT Service & Support

A comprehensive Software Maintenance program is available with SIMATIC IT products. Software Maintenance includes the following services:

- **Software Update Service** (SUS): automatic distribution of upgrades and service packs
- **Technical Support Service** (TSS): support on technical problems with SIMATIC IT software (standard support and other optional services)
- **Online Support**: a technical web site providing information such as Frequently Asked Questions and technical documentation on SIMATIC IT products

Software Update Service (SUS)

This service provides automatic shipment of new versions and service packs when released. When a new version / service pack is available for shipping, it is typically shipped within one month.

One copy of the installation DVD is shipped for each Server covered by Software Maintenance.

Hot fixes (officially tested and released) are not shipped and must be downloaded from the Technical Support Service web site.

Technical Support Service (TSS)

Siemens provides a dedicated technical support team for SIMATIC IT products.

The following options are available:

Bronze support: 9 hours/day, 5 days/week

Silver support: 24 hours/day, 5 days/week

Gold support: 24 hours/day, 7 days/week

The principal language of the SIMATIC IT hotline is English.

SIMATIC IT partners and customers covered by the Software Maintenance program are entitled to direct access to the TSS.

Access to TSS

To be able to access TSS, the customer needs to register as a user on the Technical Support web site. Connect to http://www.siemens.com/mes-simaticit/ and follow the **Technical Support Service** link.

The registration form must be completed with:

- Personal data
- The required company and plant information
- The Contract Number provided by Siemens Back Office when the contract is agreed.

Online Support

A customer who is a registered TSS user, can access the Technical Support web site (http://www.siemens.com/mes-simaticit/tss), which contains technical information such as:

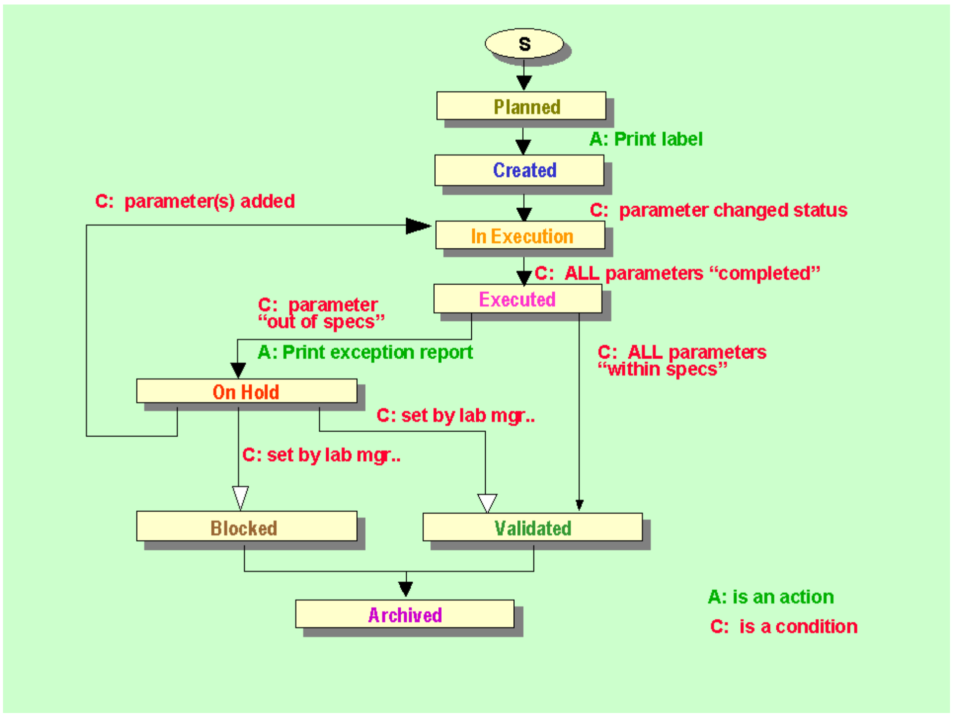
- Service conditions (Phone numbers, Working hours, Reaction times,…)
- SIMATIC IT knowledge base: a technical support database that includes practical service solutions from Technical Support or the SIMATIC IT community
- SIMATIC IT software (e.g. hot fixes, software examples) and release notes that can be downloaded
- SIMATIC IT cross-industry libraries that can be downloaded (limited access to SIMATIC IT certified partners)
- SIMATIC IT product documentation that can be downloaded
- Frequently Asked Questions and useful tips.

Table of Contents

1 Unilab Life Cycles

The life cycle model is a complete state transition diagram that specifies the possible state changes (state transitions) an object can go through.

The picture below shows an example of a typical life cycle model for samples.



1.1 Architecture

The life cycle model contains

- All possible states for a particular object
- The possible state transitions and the conditions enabling them
- Actions associated with certain state transitions for a particular object.

The list of available states is (almost) completely definable (there are just a few predefined states – and even these can be customized).

Each transition in a life cycle model has the following properties:

- An **initial state**
- A **target state**
- An **authorization list**, which defines individual users and/or user profiles who are authorized to manually trigger the state transition
- A set of **condition(s)** + **action(s)**.

Each time an event occurs, the current state of the object and the life cycle model ID are used to verify the conditions. When a condition is met, the actions are executed sequentially and the state of the object is changed.

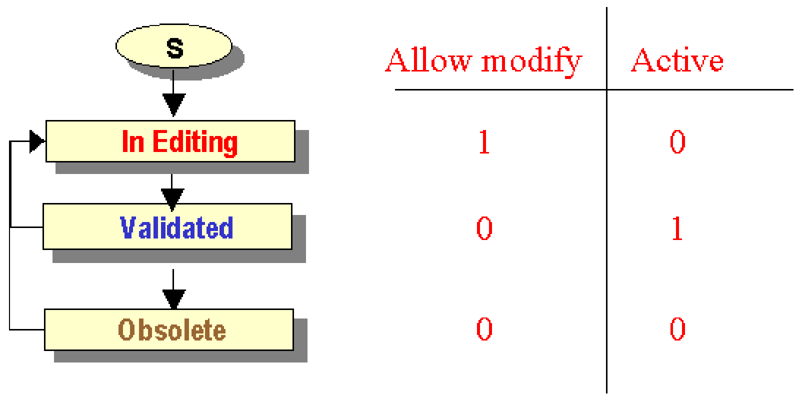
1.2 Life Cycle Definition

An ID, name, and description uniquely identify a life cycle. The life cycle ID cannot be modified once it has been saved in the database.

For each life cycle, a number of standard attributes can be set. These standard attributes can be extended with a number of user-definable attributes. An audit trail is kept for each life cycle.

To avoid recursive life cycle definitions and still have strict control over the life cycle of a life cycle model itself, the life cycle itself is controlled through the system life cycle. The system life cycle is used by default to control the configuration of the objects in the configuration model.

The picture below shows the default system life cycle.



	Allow modify	Active
In Editing	1	0
Validated	0	1
Obsolete	0	0

All state transitions are done manually. It is assumed that all users who have access to the **Define life cycle** application are authorized to make these transitions.

The status used in a life cycle model must be consistent across different life cycle models. For example, it is not possible to assign the color red to the **In editing** status in one life cycle model and assign another color to the same status in another model. All life cycles are based on a common list of states. This basic list of states can be extended at any time. Note that care must be taken when removing state definitions!

1.2.1 Life Cycle Standard Attributes

New life cycles can be created from existing life cycle templates. In the life cycle's standard attributes, the user can set whether the life cycle can be used as a template or not.

A document describing the intended use of a life cycle can be included in the life cycle definition. The document execution is implemented through the configuration of an attribute service (**unlcuse**).

For operational life cycles, the status after re-analysis can be defined. The reanalyze state is not uniquely defined (for the system as a whole – each life cycle has its own re-start entry). For one life cycle model, it could be **In editing**, whereas, for another life cycle model, it could be **In execution**. Therefore, the life cycle definition has been extended with the field **ss_after_re-analysis**. Note that this field must be filled out for each life cycle model that is applied to a method, parameter or parameter group. If the **ss_after_re-analysis** field is not filled out, no re-analysis can be performed.

Since a re-analysis is usually performed manually, it is very important to define the authorization list correctly!

1.3 Use of Life Cycles

There are no restrictions on the number of states in the model or the number of transitions between any two states. It is up to the application manager to define meaningful life cycle models. The system simply executes the instructions one by one.

Once a life cycle model is selected for a particular object, the system automatically applies it (without any manual intervention by a user). The Event Manager(s) evaluate(s) the specific conditions whenever appropriate and perform(s) the associated actions. Note that the Event Manager does not consider the authorization list; since it is a background process, it is assumed to be authorized by definition! The authorization list is applied when a user attempts to change the status manually.

When an object is created or when the life cycle of an existing object is changed, it always automatically assumes the initial status (by default). Only from that point on may the life cycle model handle any status changes in a controlled manner. Each life cycle will have only ONE default initial state, from which the first transitions can be triggered. It is not possible/supported to have multiple default entry points into a life cycle model. On the other hand, there is no specific end-point in a life cycle model.

A life cycle can (but must not) contain loops. Such a loop could even have the same begin and end status. An example of a loop is the re-analysis loop in life cycles at the operational level.

Because life cycle models are used as a reference, the **Delete** menu function is implemented in such a way that the user is discouraged from deleting a model (which could result in unresolved references). When deleting a life cycle model, the user must always confirm that he really wants to delete the model. Active life cycle models cannot be deleted.

Note that it is not feasible to automatically determine whether or not a specific life cycle model is still in use. This would require a scan of almost every table in the DB. The application manager is assumed to know which life cycles are in use.

1.4 States

1.4.1 State Definition

A state (or status) in an object's life cycle reflects one step in the procedure that should be followed to handle the object. The properties of the state define the behaviour of any object in that state. In fact, the state an object has reached defines whether this object is active and/or modifiable.

The definition of the various states is performed in the **Define states** table in the **Configuration** application.

Identification of a state

A state has a unique ID, name, and description. The state ID cannot be changed once it has been saved in the database.

Color

For each state, a color can be set that will be used in other applications to indicate that an object has reached that particular state. The color is used to display the text (the background color does not change).

Shortcut key

For each state, a shortcut key can be set to perform any state transition to that particular state. When the user hits the shortcut key, the transition is performed without displaying the **Change status** dialog.

Note

Through the ConfirmStatusChange preference, the application manager can set whether or not the Change status dialog should be displayed upon manual state transition triggered by a shortcut key. The user is then able to verify the change of status specified by the shortcut.

1.4.2 Active and Allow_modify Flag

The state of an object determines whether the object is

1. Active or not
2. Modifiable or not.

This is controlled through the **Active** and **Allow_modify** flags. The settings of both flags are included in the state definition. These flags cannot be defined/altered within a life cycle model. A particular state should always behave in the same manner, regardless of the life cycle model.

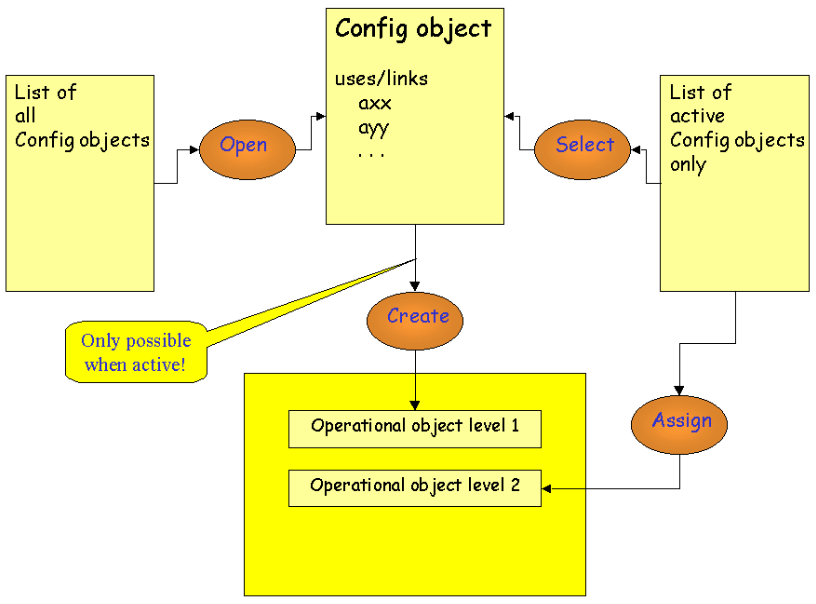
The **active** flag indicates that an object in this state is active. The interpretation of this flag depends on the object level: configuration or operational object. The **Allow_modify** flag indicates that the properties of the object in this state can be modified.

Active flag

The scope of the active flag is limited to the object to which the flag belongs. Although the **Active** flag applies to all objects, both configuration and operational, the function and interpretation is slightly different.

The configuration applications to define an object ignore the **Active** flag of that particular object. Also, to use an object definition (as part of another object - for instance, parameter profiles assigned to a sample type), both objects that are active and non-active can be used for assignment. A parameter profile that is not active can be assigned to a sample type.

The picture below shows the mechanism of the **active** flag on configuration level.

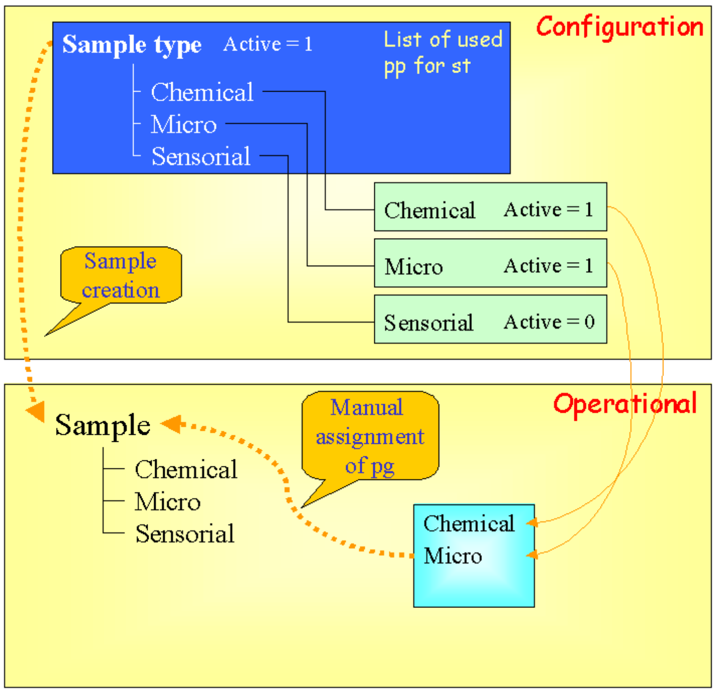

Config object
uses/links
axx
ayy
. . .
List of all Config objects
Open
Select
List of active Config objects only
Create
Only possible when active!
Operational object level 1
Operational object level 2
Assign

In the operational applications, only the configuration objects that are active can be used. No samples (or requests) of sample types (or request types) that are not active can be logged on. For example, parameter groups cannot be assigned to samples when the corresponding parameter-profile definition is not active.

Approving a configuration object (which corresponds to setting the **Active** flag), which includes the list of used objects, can be done regardless of the status of these used objects! This implies that, when the corresponding operational object is created, the list of used objects will be created as well (according to the set assignment frequencies), even if some of these used objects are currently not active.

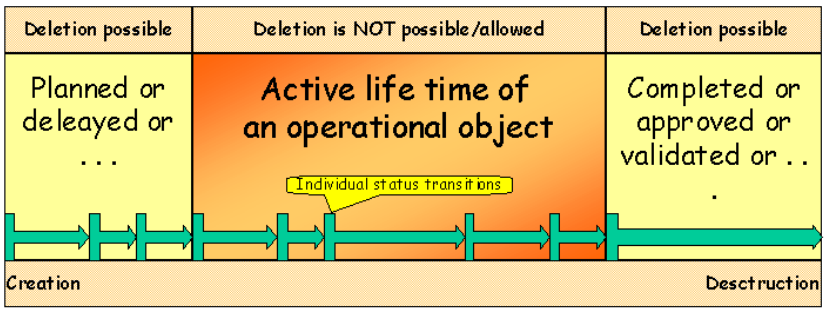
Example: When a sample is created, Unilab refers to the corresponding sample type to set the properties of that sample. When the test plan is created, the list of parameter profiles and their assignment frequencies are evaluated. The corresponding parameter groups will be assigned to the newly-created sample, even if the **Active** flag is not set for the parameter profiles. However, on manual assignment of parameter group, the pick list displays only the parameter groups of which the parameter profiles are active.

The picture below shows active versus not active objects.



For operational objects, the **Active** flag is used to indicate that the execution of the object is (still) delayed, planned, or already done. Even though the execution of a parameter, or any other sample-related object, can be delayed, the actual creation is not delayed. This should provide extra information for the analyst to prepare the delayed execution.

The picture below shows the **Active** flag in a life cycle.


Deletion possible
Deletion is NOT possible/allowed
Deletion possible
Planned or deleayed or . . .
Active life time of an operational object
Individual status transitions
Completed or approved or validated or . . .
Creation
Desctruction

Only an object that is not active can be deleted, irrespective of all other functional and data access rights! For instance, samples in Planned status are not active. Hence, planned samples can be deleted when, for some reason, they do not arrive in the laboratory.

Allow_modify flag

This status property controls whether or not the object can be modified!

At the configuration level, the **Allow_modify** flag applies to only one object level. When this flag is set for an object, the user can modify its properties, including the list of used objects. However, the properties of the used objects themselves cannot be modified.

Example: When the **Allow_modify** flag is set to 1 for a sample type, the properties of that sample type can be modified. The list of assigned info profiles and parameter profiles can be modified as well (because these are just part of the sample type object). Parameter profiles can be inserted or removed, and the assignment frequencies for the assigned parameter profiles can be modified. However, the properties of the info profiles and parameter profiles themselves cannot be modified unless their **Allow_modify** flag is set, because these are considered to be separate objects!

On the operational level, the scope of the **Allow_modify** flag also extends to the lower-level objects!

Example: When the **Allow_modify** flag is not set for a sample, it is not possible to change any of the standard properties of that sample, nor is it possible to add/remove an info card or parameter group. Likewise, it is impossibleto modify a parameter of that sample (even if the **Allow_modify** flag is set for that individual parameter).

1.4.3 Predefined system states

Some states are predefined. Note the following:

- The IDs of the predefined system states begin with @;
- Changes can be made to the characteristics of these states;
- Predefined states can never be deleted.

Default (@@)

In several life cycle models, a kind of exception handling is required. For example, it must be possible to change from any status in the life cycle to the **Cancelled** status. It would be difficult to maintain a specific transition from each status in the life cycle. To facilitate the configuration for these kinds of ‘transitions from every state’, a special kind of initial state can be defined, called **Default (@@)**. It is obvious that this special default value can be used in the **from** state field to model this requirement!

In Editing (@E)

This state is used in the system life cycle for the formal approval of configuration objects. Objects in this state are not active (**Active** flag is not set), but their properties can be modified (**Allow_modify** flag is set). Because the configuration objects in this state are not active, they cannot be used at the operational level yet.

Note that there is no restrictions against defining another edit status (independent of the @**E** status) to be used in specific life cycles.

Approved (@A)

This state is used on the system life cycle for the formal approval of configuration objects. Objects in this state are active (**Active** flag is set), but their properties cannot be modified (**Allow_modify** flag is not set). Because the configuration objects in this state are active, they can be used at the operational level.

In Test (@T)

This state is used on the system life cycle and is relevant only for systems that must comply with 21 CFR Part 11. For such systems, the properties of an approved object can no longer be modified. When trying to modify the approved object properties, a new version is triggered.

The In Test status allows testing an object on such systems. The **Allow_modify** flag is set to 0, the **Active** flag is set to 1. The system life cycle includes a transition back to **In Editing** to permit changing the object properties after testing.

It is possible to create objects in status '@T' with the Analyzer, but a specific Task has to be configured. To assume the desired behaviour, the SampleCreation task should at least contain:

- the status. A default value = '@T' can be configured

- the **version_is_current** flag. It is not possible to define a default value here, as value 'NULL' will be converted to "AND version_is_current = 'NULL'" in the where-clause. Therefore, set value_list_tp = SQL, and configure sql = "SELECT DISTINCT version_is_current FROM UTST WHERE version_is_current IS NULL".

Objects 'In Test' are not current. As a consequence, if objects are assigned to the upper level with version '~Current~' (e.g. parameter profile to sample type), they will not be assigned on the operational level (or a previous, wrong version will be assigned). Therefore, it is important to configure fixed versions on the links. After testing, this can be updated if necessary.

Obsolete (@O)

This state is used on the system life cycle for the formal approval of configuration objects. It is used to indicate objects that are no longer used. Objects in this state are not active (**Active** flag is not set), but their properties cannot be modified (**Allow_modify** flag is not set). Because the configuration objects in this state are not active, they cannot be used at the operational level.

Cancelled (@C)

This state is used at the operational level to indicate objects that are cancelled. Cancelled objects are no longer active. Canceling an object causes a trickle-down effect: all objects at a lower hierarchical level are automatically cancelled as well. When a sample is cancelled, all parameter groups, parameters, methods and info cards for that sample are also cancelled.

Planned (@P)

This state is used at the operational level to indicate objects that are scheduled, but that are not yet active. An object in this state already exists, but can still be removed (because it is - usually - not yet active). Objects in this state can be modified (**Allow_modify** flag is set).

Samples created by the **Sample Planner** always have the status @**P**.

Delayed (@D)

An object with this state already exists and will become active automatically (after the specified delay time). Objects in this state can be modified (**Allow_modify** flag is set).

1.5 Transitions

The state transitions in a life cycle define which state can be reached starting from the current state and what conditions should be met. The following types of state transitions can be distinguished:

- One-to-one state transitions
- One-to-many state transitions
- One-to-itself state transition.

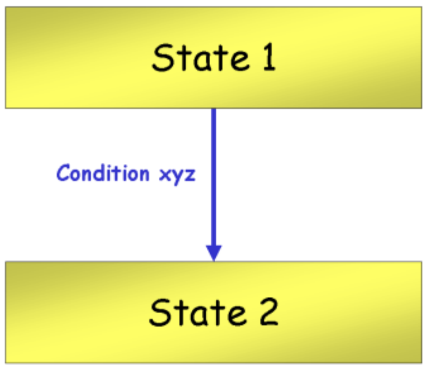
Any state transition can be triggered either manually or automatically. A user must be authorized to perform a manual transition; the condition – corresponding with the transition – is not evaluated! Automatic transitions occur only if the condition is met (= evaluates to TRUE). The authorization list is not checked!

The Event Manager evaluates the conditions that should be met to trigger the state transitions. Each state transition itself can in turn trigger one or more actions.

1.5.1 Types of State Transitions

One-to-one state transition

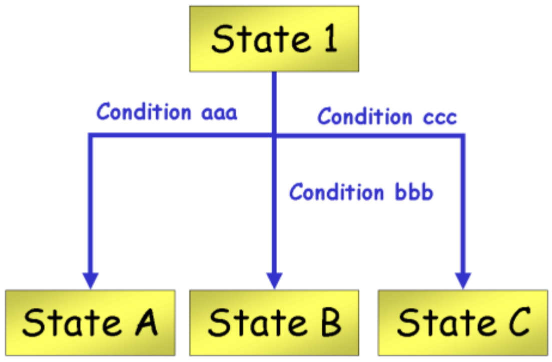
The picture below shows one-to-one state transition.



A one-to-one status transition refers to a transition in which only one status can be reached from the current status. The transition includes the condition that should be met to reach that state or the list of authorized users to trigger the transition.

One-to-many state transition

The picture below shows one-to-many state transition.



A one-to-many state transition refers to a transition in which multiple states can be reached from the current status. The status transition that actually occurs depends on the first condition that is met. Note that the order of evaluation is important; the first condition that is met (= returns TRUE) determines which branch will be taken.

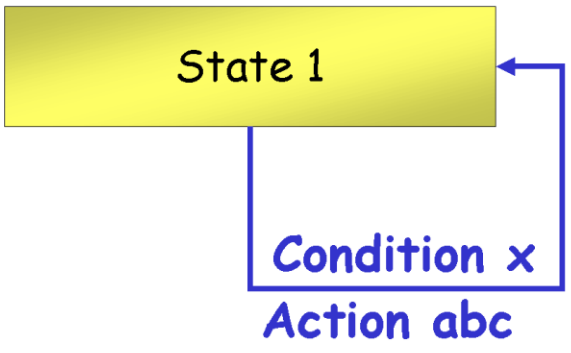
Therefore, each state transition gets a unique transition number. The Event Manager checks the conditions sequentially as controlled by the transition number. Once a condition is met, the corresponding transition is triggered: at this point, no checks will be performed on the conditions successive to the condition that was met.

Example: For a parameter in status In Execution, the following transitions can occur:

- To **On Hold**, on the condition that the result lies outside the specifications
- To **Validated**, on the condition that the result lies inside the specifications
- To **Cancelled**, a user having the authorization to do so manually triggers the transition.

One-to-itself transition

The picture below shows one-to-itself transition.



A one-to-itself state transition refers to a state transition where the begin and target status are the same. This is significant (to be used) only when an action is linked with this transition (otherwise, a **do nothing** transition would be the result).

Example: When a parameter group in the status **In Execution** detects that a specific parameter result is out of spec, an exception report is triggered, but the parameter group as a whole remains **In Execution**.

1.5.2 Manual versus Automatic Transitions

There are two kinds of state transitions:

- Automatic state transitions: Transitions that occur automatically as soon as a certain condition is met
- Manual state transitions: State transitions that are triggered manually by an authorized user.

Conditions for automatic state transitions

A condition is a custom function that describes when a state transition is allowed to occur. This function returns **TRUE** when the condition is met, and **FALSE** otherwise. The conditions used in the life cycle may refer to any property of the object or any of the related objects (e.g. the status of the parameters belonging to a parameter group). The conditions may even include user authorization.

Some examples of conditions:

- Check if a certain event has occurred
- Check if all the parameters in the parameter group are executed, in other words, verify that the status of all the parameters is **Executed**.

Note

Conditions are PL/SQL custom functions in the UNCONDITION package.

Condition for manual state transitions

The condition UNCONDITION.**OnlyManual** will often be used to prevent a specific transition from automatically occurring. This condition always returns **FALSE**, so transitions for which this condition is set are never triggered automatically.

Authorization list for manual state transitions

For each transition within the life cycle, only a limited number of users have the authorization to initiate the transition manually. This authorization feature should not be confused with the access rights specified for a particular user profile. The database administrator (DBA) is always authorized (by default - because he/she owns the entire DB and all required privileges). This means that, although the Event Manager checks the authorization, this will normally have no effect. However, if a user requests to change the status manually, then the authorization rules apply.

The authorization list defines all users and/or user profiles that are allowed to manually perform a certain state transition. When the authorized user for a certain state transition is set to **~ANY~,** all users, independent of the user profile to which they belong, are allowed to manually perform the state transition.

Dynamic user authorizations can be implemented by setting the authorized user to **~DYNAMIC~.** In this case, the authorization list is evaluated each time the corresponding state transition is triggered.

Note

The dynamic user authorization must be implemented using PL/SQL logic/code in the **UNACCESS.TransitionAuthorised()** function. The **UNACCESS.TransitionAuthorised()** function is evaluated each time a **Change…Status** function is called for a life cycle transition where the authorized user has been set to **~DYNAMIC~**.

Example: Regulatory standards for certified laboratories require that manual validation of test results be performed by a lab user having user profile Lab Management and who is not listed as a possible executor of the concerned tests. The dynamic user authorizations make it possible to check both conditions before allowing manual validation of the test results.

1.5.3 Actions

An action is a PL/SQL custom function that can be triggered when a state transition occurs.

Examples of actions:

- Changing the standard properties or attributes of the object or other objects.
- Changing the access rights of a particular object.
- Deleting a group-key entry for a method.
- Printing a label/report/certificate.
- Creating a new object depending on the object's status (e.g. trigger an analysis request when the sample material is bad or insufficient).

Note

Actions are PL/SQL custom functions in the UNACTION package.

The action is always executed as the last step of the status transition; the object is already in the new status when the PL/SQL logic is executed. This is especially important when the **allow_modify** flag is switched off (in the end status); no one, not even the DBA, can ever violate the integrity rules of the system!

Each action returns whether the function was executed successfully or not. If multiple actions are assigned to one state transition, these actions are executed sequentially. If one action fails, the successive actions are not executed.

Note

Note that all life cycle actions are executed using the DBA privileges. The following assumptions about the DBA have been made:

- The **DBA** has **up1** as the default user profile
- **up1** has always **Write** access to all objects

It is important that these assumptions never be violated, as this could affect the normal behaviour of the system!

1.6 System Default Life Cycles per Object

For every type of object (both operational and configuration), a default life cycle model can be set system-wide in the table **utobjects**. Whenever an object is created, the corresponding default life cycle is applied (but only if the creator did not specify a specific life cycle). However, an authorized user will be able to modify the life cycle model at any time.

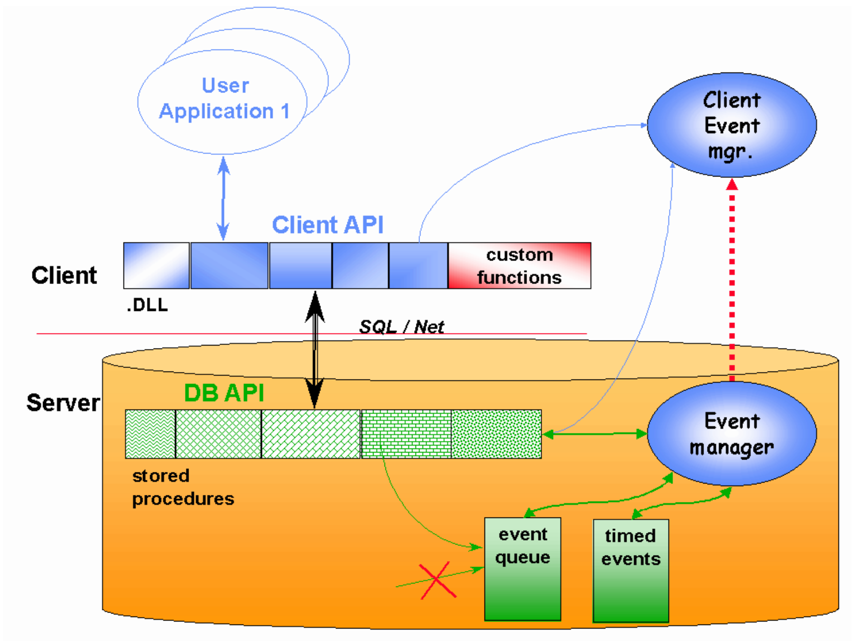
For each operational object, the default life cycle set on the system level can be overruled during object definition. For instance, the default life cycle for a sample of a certain sample type is included in the sample type definition. When this field is left blank, the default configured on the system level for the object type is applied.

1.7 Event Manager

1.7.1 Introduction

Ensuring consistency in how data is processed is a rather complex issue within a flexible environment, involving multiple workstations and users working simultaneously on multiple mutually-interacting object layers and life cycles. The Event Manager provides a consistent way of data processing.

The picture below shows the concept of the **Event Manager**.



To keep the database consistent, access to the data itself is possible only via DB API functions. These functions not only update the table in which the corresponding data is stored, but also all other data in tables that are joined to the first one. When a particular user performs an update in the database, one or several API functions are called. These APIs insert events into the event queue. The Event Manager is a server-independent job within the database, which sequentially reads and processes each of the events in the event queue.

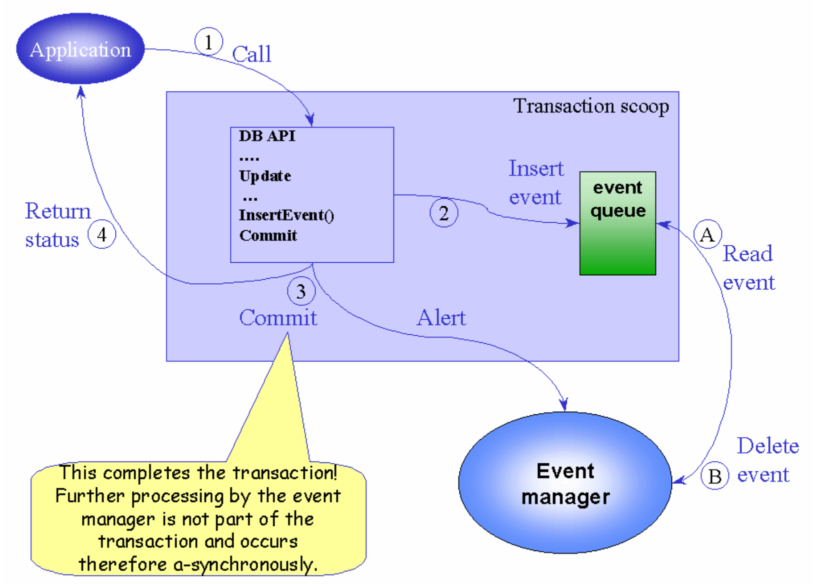
Important

The Event Manager on the database side is not able to call client program logic. Therefore, it has been extended with a **Client Event Manager** on the client side.

1.7.2 Mechanism of the Event Manager

Asynchronous event handling

The picture below shows how the **Event Manager** handles events asynchronously.



Each update operation in the database consists of a number of smaller transactions. Each transaction represents a minimal package of operations to be executed. The DB API provides the transaction control itself. In many cases, a number of APIs are processed together in one transaction. If the transaction is not completely successful, it is automatically rolled back.

Events associated with a transaction are sequentially inserted into the event table. On the **Oracle** level, the update is immediately committed and an alert is broadcast to the Event Manager. The Event Manager then processes the events sequentially, i.e. in the order they were inserted in the event table.

The client/caller side already gets the result of the database commit operation, but is not certain to what extent the Event Manager has already processed the events inserted by the API. There is a time gap between the database commit operation and the actual event processing by the Event Manager. During this time, the modify flag of the data is set to the ‘in transition’ status (presented by #), thus blocking the data from simultaneous manipulation by other transactions. If a user attempts to call an API function that could manipulate the data in transition, that call is explicitly refused by the system and the user receives an error message indicating that the data is in transition.

The Event Manager itself ignores the ‘in transition’ status of **Allow_modify** flags, since it is actually the Event Manager that performs the necessary ‘transitional’ processing to free the object for later use.

Notes

Since the Event Manager processes all events associated with database updates, the Event Manager must be running to be able to start an application.

Since the Event Manager needs to have full authority for data access and manipulation, it should always be started up with the database administrator's privileges.

Because events are handled asynchronously, it may be convenient for the user to be informed by the system when all the events have been processed. Some users may even expect the system to update the display automatically (to show the current status of each object).

In practice, the yellow DB traffic light remains on as long as the Event Manager is still processing the events of the last transaction. When the last event is processed, the traffic light changes back to green.

Auto refresh

Data is refreshed automatically after saving. This is the case for all windows, except for the main lists (sample list, method list, etc), in which it is the user who decides when to perform a Refresh. In addition, the Parameter Results window is also refreshed automatically when the method sheet details are saved (provided that the Parameter Results window is open at that moment). The Refresh operation is triggered when the yellow traffic light is switched off.

Transaction processing

In order to prevent fatal inconsistencies in the event handling process, the Event Manager applies the FIFO principle (first-in first-out).

Events associated with one transaction are processed sequentially by the Event Manager. The order in which the events are to be processed is determined by the order in which the events are inserted into the event queue.

In support of the FIFO principal, a transaction sequence number and an event sequence number is inserted into the event table. The transaction sequence identifies the transaction to which the event belongs and sets the order in which the transactions should be processed. The event sequence defines the order in which the individual events are to be processed within each transaction.

Error handling

The Event Manager is designed as a background job that should never stop running.

Therefore, it addresses an event only once, and frees the object from its 'in transition' status, regardless of whether the processing was successful or not. In this manner, incorrect processing code does not cause the Event Manager to go into endless loops. However, fatal exceptions are logged in the table **UTERROR**.

In case of debugging problems, events can be logged into a special logging table (**utevlog**). This is achieved by setting the system setting **log_ev** to 1.

Note

Given the number of events processed, one should be aware of the amount of disk space usage such a choice implies. Therefore, the system setting should be switched on only for relatively short periods of time. It should then be switched off as soon as debugging is terminated.

1.7.3 Approach to Process an Event

A large number of event types have been defined for the Unilab database. Although each event type requires specific processing, a general flow is applied to process one event (of any type). Only the processing of events involved in object creation is handled in a slightly different manner.

1.7.3.1 General Approach to Process an Event

Step 1: Life cycle evaluation

The first step in event processing consists in evaluating the current object’s life cycle. The current life cycle and status are used to evaluate what must be the new values for:

- **Status**
- **Allow_modify** flag
- **Active** flag.

Therefore, the Event Manager sequentially checks the conditions set for the different state transitions starting from the current state. The transition number in the life cycle definition sets the order in which the conditions are checked.

As soon as one condition is met, the life cycle evaluation is terminated. At this point, no checks are performed on any succesive conditions.

Step 2: Change of status

The transition that corresponds to the first condition met is triggered. This state transition is not performed using the DB API for state transition. The Event Manager directly updates the necessary fields in the database. The fields that are updated are:

- The status of the object
- The **Allow_modify** flag
- The **Active** flag.

Step 3: Execution of actions

If any actions were defined for the triggered state transition, these actions are executed in the final step. The context used for the action execution is set by the final state (i.e. the target state after state transition).

Step4: Execute the event rules corresponding to that event

The event rules (**utevrules**) that match the processed event are executed. This allows executing some actions without defining a life cycle transition. See also: Group-key assignment rules and Worklist assignment rules.

1.7.3.2 Event Processing when an Object is Created

Although, generally speaking, the same approach for event processing applies to the events involved in object creation, there do exist some minor differences.

Because newly-created objects might not yet have a life cycle assigned, the first step in the general event processing is preceded by the actual life-cycle assignment. All transitions starting from the initial state can be evaluated only after life cycles have been assigned.

Upon object creation, the group-key assignment rules are evaluated (if relevant). Note that, for methods, the result of the evaluation of the group-key assignment rules can be overruled by the worklist assignment rules in a later step of the event processing.

The object access rights are also initialized (by calling the **UNACCESS.InitObjectAccessRights()** PL/SQL function).

1.7.4 Primary and Secondary Events

Actions associated with a life cycle state transition can call a DB API, thus causing the insertion of a number of (additional) events in the event table. The Event Manager also processes these secondary events. The ‘in transition’ state of the corresponding object’s modify flag is freed by the Event Manager only after the entire set of secondary events has been processed.

Although secondary events are not part of the original transaction (because the original transaction has already committed and returned control to the caller), they are inserted into the event queue with the original transaction number. This is necessary to keep all events logically together when multiple Event Managers are running in parallel.

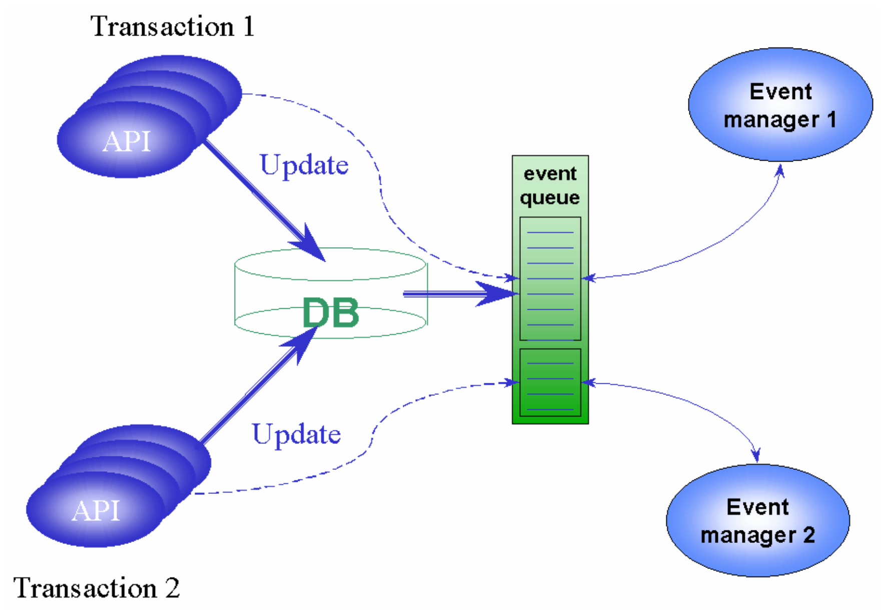
1.7.5 Multiple Event Managers

The Event Manager processes all events associated with a particular transaction sequentially. The Event Manager only starts processing the events associated with a transaction once the event processing of the previous transaction has been fully completed. Given the number of events that can be associated with one transaction, this might affect performance and responsiveness.

Therefore, the event processing speed can be dynamically balanced by working with several Event Manager jobs at a time. Each instance will process the events associated with a specific transaction, but in parallel (thus speeding up the process of event handling when multiple transactions occur in parallel).

In fact, when an event is inserted into the event table, an alert is broadcast, waking up all inactive Event Manager jobs. The first job able to access the event table selects and processes the events connected with the first transaction. As long as unallocated transactions remain available within the event table, individual Event Manager jobs can allocate one of these and process them in parallel with the other Event Manager jobs. When the event list is finally empty, the Event Manager jobs return to the sleeping mode.

The picture below shows the parallel processing of events from different transactions by multiple Event Managers.



The number of Event Managers can be set in the system setting **event_mgrs_to_start**.

Note

Each Event Manager consumes a part of the server memory and requires a certain amount of disk space in the database. Therefore, it is recommended not to exaggerate the number of Event Managers. For reasons stated above, two Event Managers are necessary. For more then two Event Managers, the following rule of thumb can be applied:

- From 30 concurrent Unilab users: Third Event Manager
- From 60 concurrent Unilab users: Fourth Event Manager.

1.7.6 Trickle-up and Trickle-down Effect

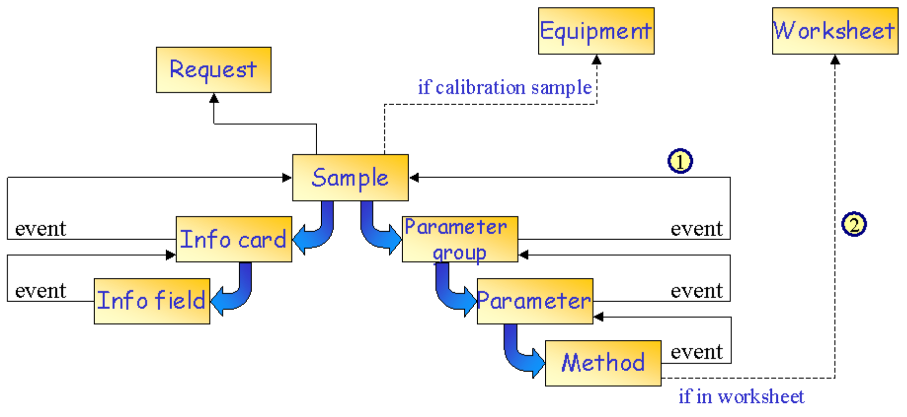
Events associated with a transaction are DB API-specific. Moreover, these events are to be handled within the context of a specific object. On the other hand, the Unilab database is organized in a strictly hierarchical manner. Events processed on a certain level in this model will unavoidably have an effect on higher and/or lower hierarchical levels.

In other words, the event handling process has to perform object-specific event handling within the context of a hierarchical data base model with mutually interacting objects. This problem is solved by implementation of the trickle-up and trickle-down effects. These effects simplify the actions and conditions in the concerned life cycle models.

Trickle-up

The trickle-up effect refers to the mechanism that also propagates a single event to the parents of the original object. When, for instance, a method is updated, the parameter life cycle must be evaluated as well. This life cycle evaluation could lead to a change of state of the parameter. The parameter-group life cycle also must be evaluated, and so on. An event on the method level thus causes a cascade of life cycle evaluations on the higher hierarchical levels.

The picture below shows the concept of trickle-up.



An event processed on a certain level in the data model will be passed to the higher hierarchical level to be evaluated there as well. For the event that is passed to the next level, a number of fields are replaced. The object ID and object type are replaced by the appropriate value on every level. The life cycle and the current status of the object are re-evaluated as well on every level.

The trickle-up effect is continued up to the highest hierarchical level. In many cases, the highest hierarchical level corresponds to a sample. In case the sample belongs to a request, the trickle-up effect is continued up to the request level. Note that the request must be filled out in the sample standard properties.

In case the sample is an intervention sample, the trickle-up effect is forwarded to the equipment itself.

For methods belonging to a worksheet, the trickle-up effect is forwarded to the worksheet as well. However, events triggered on the method level are only propagated to the worksheet level in a second trickle-up operation after the trickle-up effect up to the sample level (or request level) has been terminated. In this manner, the evaluation of the worksheet life cycle can be made dependent on the sample to which the method was assigned.

Trickle-down

The trickle-down effect refers to the mechanism of handling a cascade of status changes at lower hierarchical level(s) in one single transaction. This trickle-down effect is only implemented/required for the cancel and re-analysis APIs.

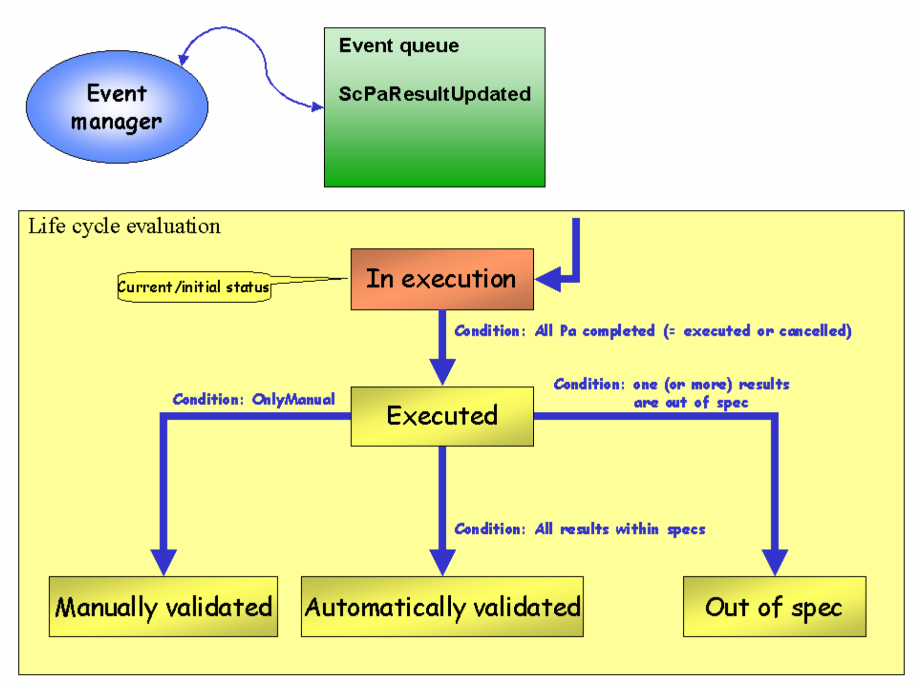
When an object is cancelled (by changing its status to @**C**), all objects on lower hierarchical levels are also cancelled by default. When, for example, a parameter is cancelled, all underlying methods are also cancelled. Objects in the status **Cancelled** can no longer be modified. Without the trickle-down effect in place, the Event Manager could no longer cancel these methods, because the parameter would no longer be modifiable.

1.7.7 Using InsertEvent in Actions

Actions associated with state transitions in a life cycle can, in principle, call any DB API. However, there are a few exceptions to this general rule. The **ChangeStatus** API is one of the prominent APIs that cannot be called directly from within an action. The reason is that an action is linked/associated with a specific transition in the life cycle; the **ChangeStatus** API would create a very ambiguous state because it would create a new transition while the current transition is still in progress.

However, an alternative way of working is provided by the **InsertEvent** mechanism. The action inserts an event that forces the state transition (after this transition has been completed).

The picture below shows an example of a (simple) life cycle of a parameter group.



When the last parameter result is filled (it does not matter how this is done – manually or through a calculation), the **ScPaResultUpdated** event will be evaluated and the status will change from **In Execution** to **Executed**. That is the end of the story: the event is handled!

The **Automatically validated** status will only be reached the next time an event is generated that triggers the evaluation in this parameter group life cycle. Modifying the **PaCompleted** condition as follows can solve this:

```
BOOL PaCompleted = TRUE;     -- we start optimistically
BOOL AllResultsWithinSpecs = TRUE ;
BOOL ResultsOutOfspecs = FALSE ;
FOR EACH pa IN pa_list_of_this_pg
     IF NOT ( pa.ss IN (“Executed”, “Cancelled”) ) THEN
          PaCompleted = FALSE ;
     END IF
     IF ( pa.value_f IS NULL ) OR ( pa.ss <> “Executed” )
THEN
          AllResultsWithinSpecs = FALSE ;
     ELSE IF pa.value_f is out of spec THEN
          ResultsOutOfspecs = TRUE ;
          AllResultsWithinSpecs = FALSE ;
     END IF
NEXT pa
IF AllResultsWithinSpecs = TRUE THEN
     ev_details = "All results within specs"
     InsertEvent ('<ThisAction>', UNAPIGEN.P_EVMGR_NAME,
          object_tp, object_id, object_lc,
          object_ss, ev_tp, ev_details, seq_nr ) ;
ELSE IF ResultsOutOfspecs = TRUE THEN
     ev_details = "One (or more) results are out of spec"
     InsertEvent('<ThisAction>', UNAPIGEN.P_EVMGR_NAME,
          object_tp, object_id, object_lc,
          object_ss, ev_tp, ev_details, seq_nr ) ;
END IF
```

Note that the actual conditional logic has been completely moved to the condition linked with the **In execution – Executed** transition. The condition on the **Executed – Automatically validated** transition can be simplified to

```
ev_details = "All results within specs"
```

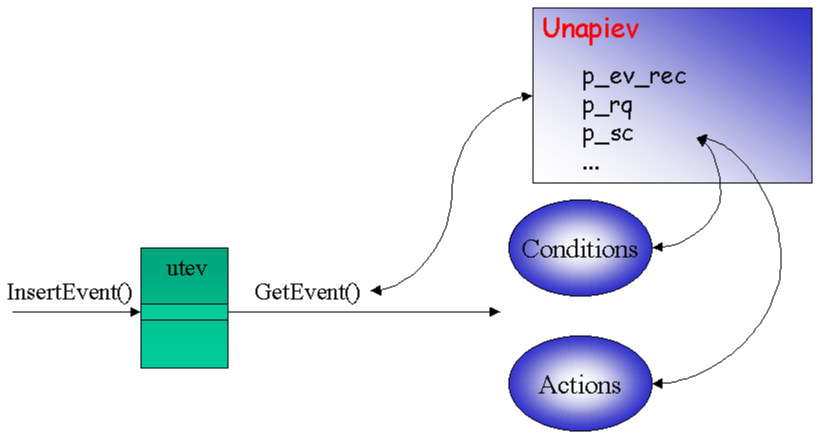
Note also that the **InsertEvent** mechanism can also be used to send events to other objects!

1.7.8 Event Manager Environment Variables

Actions and conditions are PL/SQL custom functions that require the input variables for proper execution. These input variables are defined by the context in which the conditions or actions are called (the sample code, the parameter group, parameter, etc).

The context in which an event is called is kept in a number of specific variables defined in the **UNAPIEV** package in the database. The event itself is referenced by the variable **UNAPIEV.P_EV_REC**. The type and number of context variables depends on the context in which the event is called.

The picture below shows the Event Manager environment variables.



1.7.9 Timed Events

There are also events that must be executed at a specific moment in time. When the moment of execution arrives, the event is moved from the timed event queue to the actual/normal event queue.

Examples of timed events are:

- Implementation of delay for parameter groups and parameters: the active flag of the objects should be set after a certain period of time has elapsed.
- Timed calibration rules

Timed events can also be created (using a DB API function) from within:

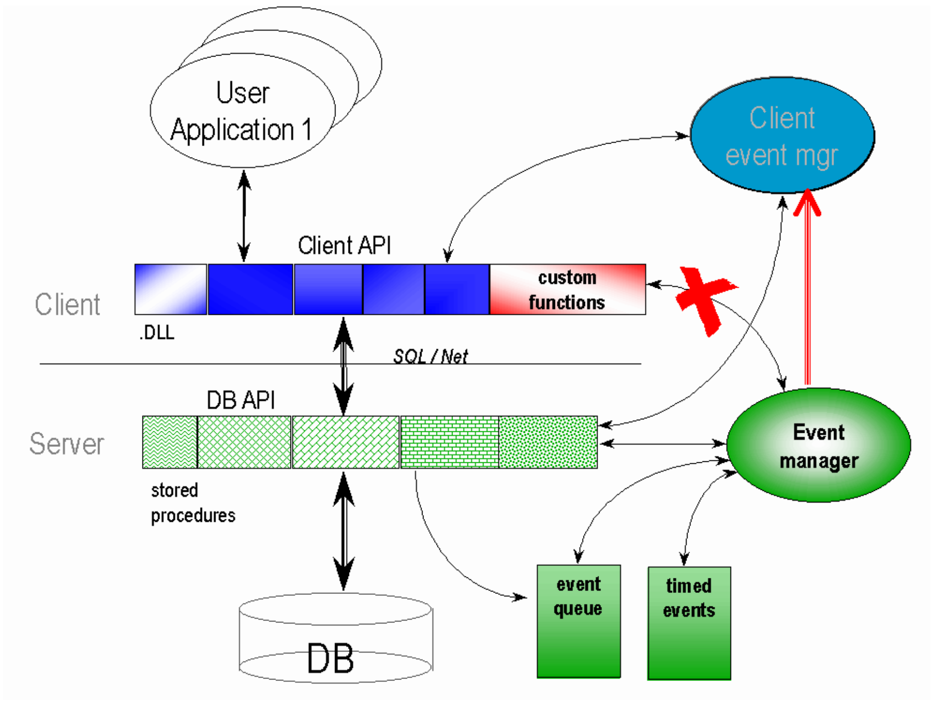
- A specific action in a life cycle model
- The DB API that handles the creation of delayed objects

Timed events are stored in a special event table, **utevtimed**. This table contains supplementary information as to the date and time when the event must be processed.

1.7.10 Distributed Event Manager

The database Event Manager is not able to execute program logic running on the client (such as custom VC++ code). This is why the concept of the **Client Event Manager** has been introduced.

The picture below shows the **Client Event Manager.**

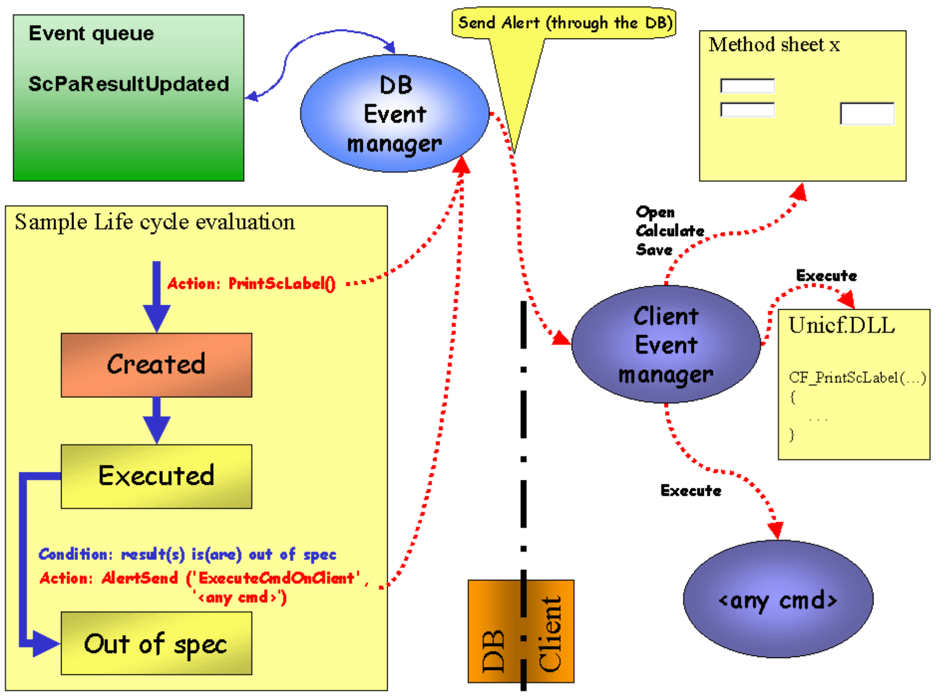


About the Client Event Manager

The **Client Event Manager** runs in the background (i.e. no user interface is provided). Actually, it is permanently waiting for an alert from the database Event Manager to execute a specific task.

As shown in the picture below, some functions of Unilab run on the server side, whereas others run on the client side. Functions, running on the client side are grouped into **DLL**s. The database Event Manager is not able to execute custom functions on the client side. However, it is able to send an alert to the **Client Event Manager** to have the appropriate custom function executed by the latter.

The picture below shows the increased flexibility with the **Client Event Manager**.



The **Client Event Manager** contributes significantly to the flexibility of the system. To be more precise, the **Client Event Manager** is able to:

- Recalculate a method sheet whenever one or more cells have been updated.
- Support the triggering of the label printing function (for samples and requests) from within the database (action triggered by a transition in the object's life cycle).
- Execute any Windows command.

1.8 Electronic Signature

An electronic signature creates an audit trail / evidence (to a specific object within the system) that the user has performed a certain action on the object and that the object has been correspondingly modified. To determine the identity of the user (before this evidence can be created), the userID and password must be entered in the same window that describes the action the user is signing off.

In Unilab, a list of states that require an electronic signature can be defined. This is done through the system setting **States_to_sign_off**. Each time an authorized user initiates a transition to such a state, he/she will be forced to enter their user ID, password and a comment on the state transition. When the signature is found to be valid, an entry will be saved in the history of the corresponding object (even if history logging was switched off).

2 User-Defined Attributes

User attributes are user-definable properties that allow expanding the definition of an object in order to capture additional information. Any object can have additional user-definable attributes.

Attributes are very versatile objects with quite a range of different applications. Attributes can, for instance, be used for:

- Representing key information for reporting purposes (e.g. ISO method name, commercial name of the sample type or the translation of an object name into another language).
- Implementing site customization: site-specific properties can be added to the definition of standard Unilab objects.
- Making other files such as **Word** documents, **Excel** files or pictures accessible from within Unilab.

For this last category of attributes, the application used to access the files is contained in the attribute definition as a service. These attributes are defined on the system level; they do not have to be assigned to a specific object.

2.1 Defining Attributes

An attribute definition consists in a name, a description and a list of allowed value(s). For attributes that are used to activate/execute an external application, the service (the application) that opens these files must be defined.

A default value

For each user-definable attribute, a default value can be specified. This value can be overruled by assigning the attribute to a configuration object.

List of values

An attribute definition consists in a name and a list of value(s). The following lists of allowed values are supported:

- Existing values: the selection list for the attribute’s value contains values that have already been assigned.
- Fixed list: the user enters a fixed list when defining the attribute.
- Dynamic list: the list of values will be defined by executing an **SQL** statement.

Note that the attribute definition can be configured so that the user is allowed or forbidden to enter a value that is not in this list.

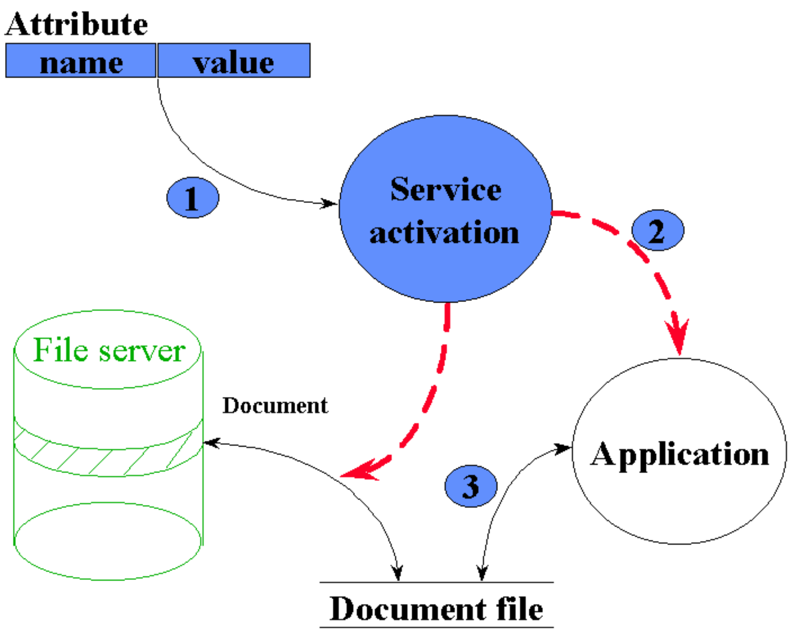
Attribute services

An attribute can have a service associated with it. The attribute service is an application that is used to display or access the file contained in the attribute's value. For example, the service can be **MSWord** to open a standard operating procedure (SOP) attached to a sample, **Excel** or any other (client) application. This approach can, for instance, be used to make a chromatogram, related to a specific method sheet, easily accessible from within the method sheet itself.

The following notation is used as a placeholder for the actual attribute value: **~value~**

The service itself can be an application that allows modifying the document (for instance, MS WORD), or an application that only allows reading the document (such as WORDVIEWER).

The mechanism of an attribute service is displayed in the picture below.



An attribute is defined and includes a reference to the service to be activated. This attribute is assigned to the appropriate object (e.g. a sample type). The attribute value is set to the name of the file that needs to be activated. This file is stored on a file server. When the attribute is launched:

3. The service is activated.
4. The application (for instance, a document editor) is opened.
5. The appropriate file is displayed.

Note

To make sure that document files can be accessed from any client, these files must be available on a shared file server.

Example: A method may not consist in only a set of data values, but can also refer to a graph (from which these data values have been extracted). Most likely, there will be an application to control the instrument that generated this graph. Assuming this graph is saved in a file, then the attribute list related to the method could be:

Attribute name	**value**
...	
graph name	<graph file name>

When the attribute is launched, the system will run the application as defined in the properties of the attribute **Graph name** and display the details.

An attribute can have a shortcut key assigned. This allows the user to call the service for the appropriate attribute from different places in the other applications with a single keystroke. The mode in which the service must be opened (minimized, maximized) can be defined as well.

2.2 Special Service Attributes

Some standard system functions are implemented by means of special service attributes. The attribute definition then includes the application used to automate the process. Another characteristic of these attributes is the fact that they are defined on the system level; they do not need to be assigned to each specific object.

The processes that can be implemented in this manner include:

- Online availability of SOPs and descriptive documents
- Online trending
- Report menus
- Predefined comments

SOP and descriptive documents

For several objects, standard operating procedures or a descriptive document can be made available on line. The name of the corresponding file can be defined in the objects' standard properties. The application used to display the SOP is set as the service in an attribute. The attribute name is usually **un<xx>doc**, where **<xx>** refers to the object. The object for which a SOP or a descriptive file can be defined and the corresponding attributes are listed in the table below.

Object	**Attribute name**
Sample type	Unstdoc
Sample code	Unscdoc
Request type	Unrtdoc
Request	Unrqdoc
Method sheet	Unmtdoc
Operational method	Unmedoc
User profile	Unupdoc
Life cycle use	Unlcuse
Equipment usage	Unequsage
Equipment accessories	Uneqaccessories
Equipment operational Details	Uneqopera
Sample type	Unstdoc
Sample code	Unscdoc

Report menus

From within the **Report** menu of the **Configuration** application and the operational application, the user can select from up to 10 different reports to be activated. The **Report** menus are active only when there is a specific object selected (otherwise, it is not possible to pass the object context information)! When selected, the report is opened and refreshed for the object (sample type, sample code, etc.) that had focus at the moment of report selection. This process is implemented by means of a special service attribute.

The application used to generate the report is set in the service attribute. The reports that correspond to these menus are customized by means of special service attributes. These service attributes are named **un<yy>report<x>** where **<yy>** refers to the main object and **<x>** refers to the menu entry. For instance, when the menu **Report 3** is activated in a sample list, Unilab looks for the **unscreport3** attribute definition to determine the report that is actually used.

Predefined comments

Whenever a comment must be entered, a user can make a selection from a dialogue listing a set of predefined standard comments. The challenge in configuring standard comments lies not in just defining one general list of comments used for all objects. Instead, a list must be defined, not only for each object type, but even per object ID. Indeed, the recurring comments for a sample are usually quite different from those that are used for a method. In addition, it is often required to have different picklists per product type (= per sample type).

The predefined standard comments are implemented by means of special service attributes. These attributes have a name like **Uncomment<obj_tp>.** The **<obj_tp>** refers to object type (**ST, SC**, etc.). The service attribute accessing the standard comment for samples is named **Uncommentsc**.

The list of values defined for the service attribute is used to establish the list of comments displayed in the **Comments** dialogue. This list of values can be a fixed list or a dynamic list. By using a dynamic SQL statement, the object ID **<obj_ID>** is used to resolve the current context.

Example:

In order to get a sample type specific list of standard comments in the comment dialogue of a sample, the application manager must define the **Uncommentsc** attribute. The list of values for the attribute is defined as:

```
SELECT std_comment FROM TABLE utcomment
WHERE comment_group = '~sc@st~'
```

2.3 Handling User-Definable Attributes

Objects on the operational level can inherit the attributes assigned to their corresponding configuration objects. For example, a **Word** document defined as an attribute for a sample type can be inherited by all samples of that sample type. The document is then also available on line at the operational level.

This paragraph describes the rules that apply for the inheritance of user-definable attributes.

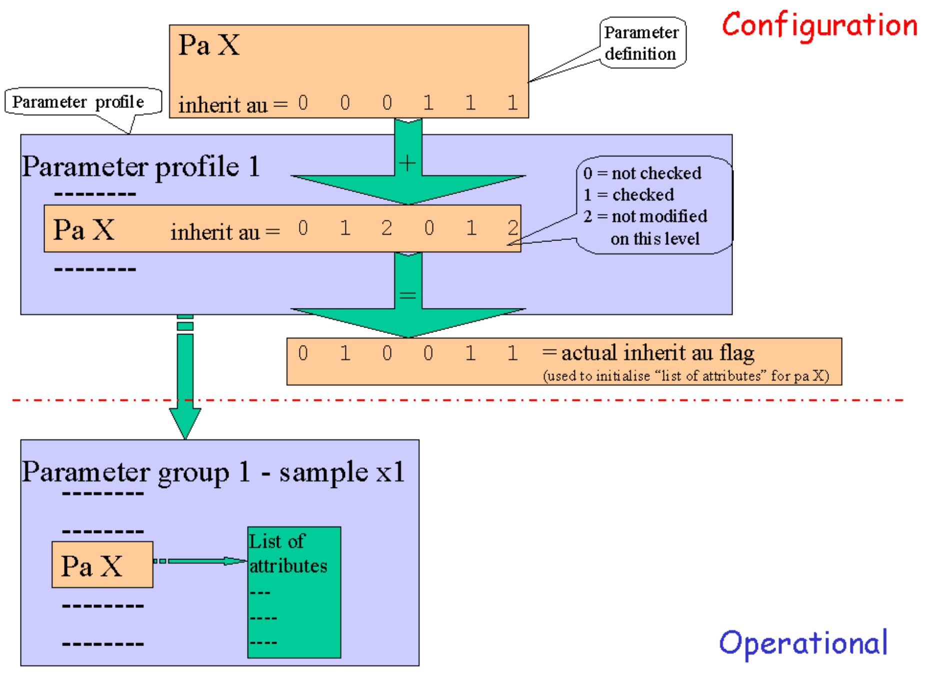
Note

This concept is not directly related to the definition of an attribute. It is explained here because it is applicable to any object that has user-definable attributes.

2.3.1 Inherit Attributes Flag on Object Level

On the configuration level, the **Inherit attributes** flag can be specified for different levels. In some cases, it is a two-state checkbox; in others, it is a tri-state checkbox.

The picture below shows the meaning and function of these flags for an operational object. Note that the parameter profile is used merely as an example to illustrate the general principle (which is applicable to any object level in the configuration part of the DB).


Configuration
Parameter definition
Pa X
inherit au = 0 0 0 1 1 1
Parameter profile
Parameter profile 1
+
0 = not checked
1 = checked
2 = not modified on this level
Pa X inherit au = 0 1 2 0 1 2
=
0 1 0 0 1 1 = actual inherit au flag
(used to initialise "list of attributes" for pa X)
Parameter group 1 - sample x1
Pa X
List of attributes
Operational

In the case of an attribute assigned to a parameter (**Pa X**), the flag can be set both in the parameter definition itself, or on the link between the parameter and the parameter profile (the pp-pr link).

Inherit attributes on pp-pr link

Basically, the value for the inherit flag on the link parameter–parameter profile (pp-pr link) overrules the value on parameter level. Only when the value is not set on the pp-pr link are the parameter definition settings applied.

If the inherit flag is set to 1 on the pp-pr link, the attribute list for the parameter on the operational level is initialized as follows:

- First all attributes from the link parameter–parameter profile (pp-pr link) are assigned;
- Then all attributes from the parameter definition that are not yet assigned are added.

Inherit attributes on parameter level

Alternatively, when the actual inherit flag is not set on the pp-pr link, the inherit flag of the parameter definition is checked. Only when this flag is set to 1 are the attributes assigned on the parameter definition level inherited by the parameter on the operational level.

General assessment

In either case, the inheritance of attributes by the operational objects must always be considered very carefully. Note that when the **Inherit attributes** flag is set on the configuration object level, the entire set of attributes and values is always inherited. When configuration level attributes have a static character (= do not change), it makes no sense to duplicate them several thousands of times for each single operational object!

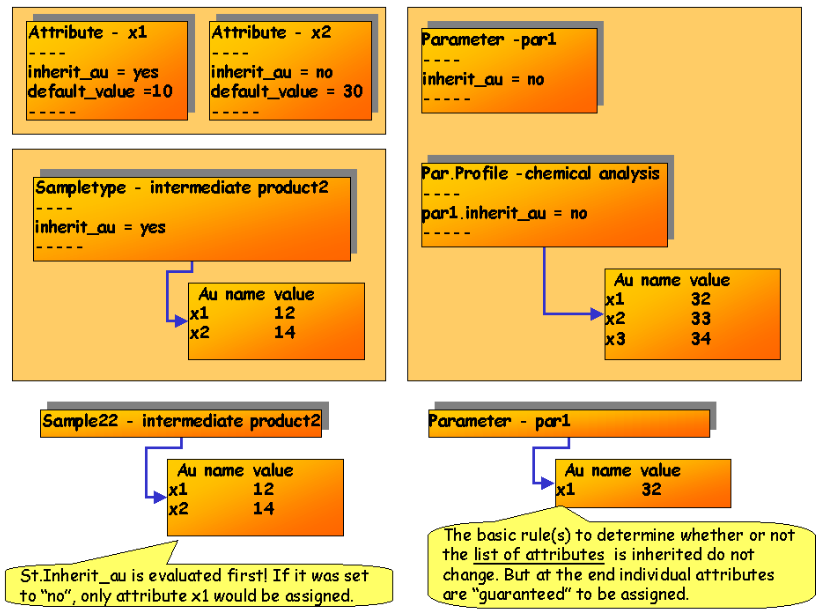
An elegant solution to this problem is provided by the fact that attribute values can be used as the default value for an info field. The value of the sample type attribute **Commercial name** can be used as the default value for a sample info field (that can carry the same name or not). This eliminates the need for inheriting all sample type attributes on the operational level.

Another solution consists in setting the **Inherit attributes** flag within the attribute definition itself. This way, it is possible to specify for each attribute individually whether it has to be inherited on the operational level or not.

2.3.2 Inherit Attributes Flag on Attribute Level

The definition of a user-definable attribute includes a flag to specify whether or not this specific attribute must be inherited by operational objects from their corresponding configuration objects. The setting of this property implies that the attribute is always inherited, regardless of the configuration settings on the object level.

The picture below shows the **Inherit attributes** flag on the attribute level.


Attribute - x1
----
inherit_au = yes
default_value =10

Attribute - x2
----
inherit_au = no
default_value = 30

Parameter -par1
----
inherit_au = no

Sampletype - intermediate product2
----
inherit_au = yes

Au name value
x1 12
x2 14
Par.Profile -chemical analysis
----
par1.inherit_au = no

Au name value
x1 32
x2 33
x3 34
Sample22 - intermediate product2
Au name value
x1 12
x2 14
Parameter - par1
Au name value
x1 32
St.Inherit_au is evaluated first! If it was set to "no", only attribute x1 would be assigned.
The basic rule(s) to determine whether or not the list of attributes is inherited do not change. But at the end individual attributes are "guaranteed" to be assigned.

The **Inherit attributes** flags on the object level are always evaluated first. This guarantees the basic rules still to be valid. However, when the actual **Inherit attributes** flag on the object level returns **No**, the corresponding setting on the attribute level is evaluated. If this flag returns **Yes**, the attributes are inherited.

2.3.3 Example

For an attribute assigned to a sample type, the following combinations can exist for the **Inherit** flag:

AU	0	1	0	1
ST	1	1	0	0
Inherited ?	1	1	0	1

If the attribute is assigned on the link **RTST**, the number of possible combinations for inheriting the attribute increases:

AU	0	1	0	1	0	1	0	1	0	1	0	1
ST	1	1	0	0	1	1	0	0	1	1	0	0
RTST	0	0	0	0	1	1	1	1	2	2	2	2
Inherited ?	0	1	0	1	1	1	1	1	1	1	0	1

3 Audit Trail

An audit trail is kept and maintained for each object. Audit-trail logging keeps a full audit trail for the object, thus supplying answers to the following questions:

- What happened to the object?
- When did the change take place?
- Who was responsible for the change?
- Why was the change made (optional)?

If no audit trail is required, this option can be switched off. However, the following items will always be logged:

- The fact that the audit trail was switched off
- Object-specific comments that were explicitly entered
- Modify reasons in case of re-analysis, cancel of objects, electronic signature, manual change status and change life cycle

The GUI does not have a command to delete the audit trail of a particular object. However, the life cycle model of the corresponding object can handle this. When an object is archived, it is straightforward to attach the action to clean up the object's audit trail.

The switching on/off of the audit trail can be controlled automatically by way of actions in the life cycle model. This is especially used for configuration objects: during the creation of a new object, tracing all changes is not relevant. Once the object becomes operational (i.e. is approved), any changes (usually) must be traced!

When deleting an object on the configuration level, the references to this object (from within a higher-level configuration object) are also deleted. When an info profile is deleted, the references to that info profile from within (a number of) sample types are also deleted (otherwise, deleting the info profile would result in an inconsistent database). However, the audit trail of these sample types shows the deletion of the info profile (which is the only trace of the existence of such an info profile). Please note that this is not implemented for all deletions of all object types. When deleting an attribute definition, that attribute will not be deleted from all objects using that attribute. Such an operation would last hours on operational systems containing a lot of samples.

The table below lists the typical **What** (has been done to the object) categories that are supported. Note that messages on the configuration level are generic. On the operational level, messages include reference to the object type (on the operational level, **Object** is replaced by the object name or the object abbreviation). For instance, when the standard properties of a sample type (configuration object) are updated, the message **ObjectUpdated** is entered in the audit trail. When the standard properties of a sample (operational object) are updated, the message **SampleUpdated** is entered.

The table below lists the statements for history logging.

What	Comment
ObjectCreated	Always the first entry in the object's history. Even when the logging is switched off by default, the **ObjectCreated** is inserted in the history!
EvMgrStatusChanged	Is used to indicate a change of state initialized by the Event Manager (automatic state change)
ObjectStatusChanged, ObjectReanalysis, ObjectCanceled	Is used whenever the state of an object is changed manually. The system automatically fills out the **Why** field with the default text: <Old State> => <New State>
ObjectLifecycleChanged	Is used whenever a new life cycle model is used to monitor the object’s state changes. This entry is suppressed when the sample is created. The system automatically fills out the **Why** field with the default text: <Old lc> => <New lc>
ObjectUpdated	Is used whenever the standard attributes of an object (except the state and life cycle reference) have been modified.
UsedObjectsUpdated	Is used whenever the list of used objects is modified (i.e. used objects are added or deleted).
Comment	Used when an additional comment is entered.
Archived	Whenever an archive is made of an object and the object is not removed from the DB, this entry is added to the object’s history.
AttributesUpdated (<xx>AttributesUpdated)	Is used whenever an object's attributes list is modified.
AccesRightsUpdated (<xx>AccessRightsUpdated)	Is used when the object’s access rights are updated.
<xx>Cancelled	Is used whenever an object is cancelled (<xx> refers to the object abbreviation). This statement is not applicable on the configuration level.
History switched OFF	Last entry in the history when history is switched off.
History switched ON	To register the date and time when the history logging was started again.
ElectronicSignature	Always associated with another action in the audit trail that has been signed electronically. There may be a comment on the signature.

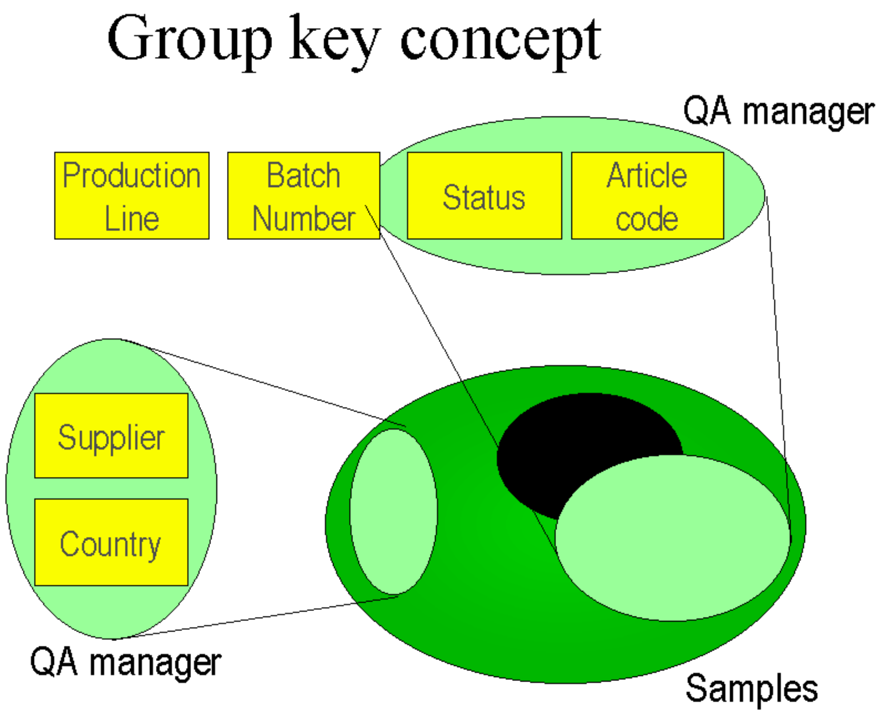
There are 2 possible descriptions for the “who” (**who** and **who_description**) and the “what” (**what** and **what_description**) that are maintained by standard Unilab. The layout for the audit trail may be customized according to one of the 2 available descriptions. The **Who** column contains the userID (**ZW333456**) that is used to connect on the system, the **who_description** will contain an explicit user name (**Glenn Taylor**). The **What** column contains a short description that can be interpreted by people knowing the configuration (e.g. **ObjectStatusChanged)**, the **What_description** contains a more user-friendly text that can be used as such in a report (e.g.: status of sample type **Tomato soup** is changed from **Approved [@A]** to **In Editing [@E].)**.

4 Group Keys

4.1 Selecting Data from the Database

A key issue in LIMS design consists in obtaining maximum flexibility while keeping both processing time and data storage requirements under control. The group key concept provides a powerful example of how this objective was reached within the area of data selection.

The picture below shows the concept of group keys.



The solution to achieve the best possible performance in data selection is to use an indexing mechanism. The organization of data in a phone directory illustrates the mechanism and advantages of index-based data selection. The reader is first guided to the subset of all phone subscribers located in a specific region or city (index 1). Within this subset, he/she systematically scans through the alphabetically-ordered family names (index 2); as long as necessary, the reader is guided systematically to more restricted subsets (subscriber's initials - index 3, street - index 4, etc). Guided in the search by a 'concatenation' of indices, the reader finds the phone number within seconds, whereas the same search in a randomly-organized (non-indexed) list might require at least several hours.

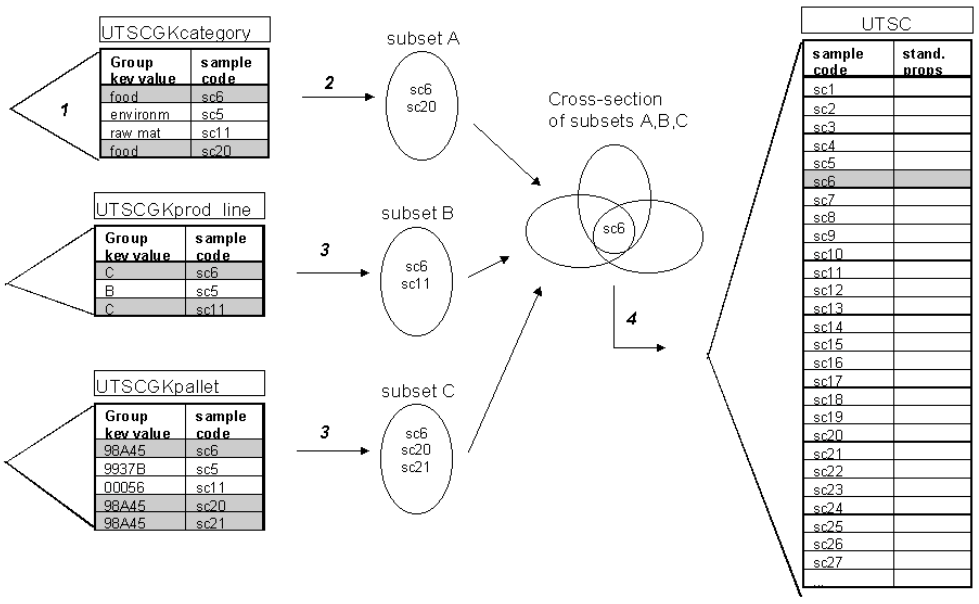
In many cases, only a small subset of the samples requires a specific key. For example, it is only necessary for the raw materials to know the supplier of the product; this key is not relevant/applicable for final products or environmental samples. Therefore, the concept of partial indices has been introduced (because this can drastically improve performance).

4.1.1 Selecting Object Sets By Using Group Keys

The system supports group keys for the following major database objects:

- Request types & Requests
- Sample types & Samples
- Methods
- Worksheets
- Protocols & Studies

Each of these types of group keys is implemented using the same principles and technology. The picture below shows the principle for using several group keys (in random order) as a combined key to retrieve a specific set of samples.


UTSCGKcategory
Group key value | sample code
food | sc6
environm | sc5
raw mat | sc11
food | sc20
1
2
subset A
sc6 sc20
UTSCGKprod_line
Group key value | sample code
C | sc6
B | sc5
C | sc11
3
subset B
sc6 sc11
UTSCGKpallet
Group key value | sample code
98A45 | sc6
9937B | sc5
00056 | sc11
98A45 | sc20
98A45 | sc21
3
subset C
sc6 sc20 sc21
Cross-section of subsets A,B,C
sc6
4
UTSC
sample code | stand. props
sc1 sc2 sc3 sc4 sc5 sc6 sc7 sc8 sc9 sc10 sc11 sc12 sc13 sc14 sc15 sc16 sc17 sc18 sc19 sc20 sc21 sc22 sc23 sc24 sc25 sc26 sc27 ...

For each group key, a separate table is created in the database, containing two indexed columns:

- One column contains the sample identifier (sample code) of each of the samples that have actually been assigned a value for the given group key
- The other column contains the assigned group-key value.

The lab processes many sample categories. All **Food**-related sample types are assigned the group-key name **category** and group-key value **food** on sample creation. As the info fields **product_line** and **pallet** are filled out, info values are filled into the corresponding group keys (linked/identified by the same name). The data selection request is handled in the following manner:

- Indexed search for all entries in the **Category** group-key table with the value **Food** using **UTSCGKcategory** as a lookup table yields a subset A of sample codes
- Step 1 is repeated for the group keys **prod_line** and **pallet**. This results in subsets B and C of sample codes
- An indexed search is performed on the main sample table (**UTSC**), referring to the cross-section of the subsets A, B and C

4.1.2 Finding Group-Key Information for Specific Samples

The set-up presented in **Selecting Object Sets By Using Group Keys** works perfectly well if the query is conducted in the manner described: i.e. using keys to locate samples. However, the table structure shown therein does not offer good performance when the query is conducted to find the opposite: i.e. to find all group keys and values for a specific sample. Indeed, in such a case, as many tables would need to be searched as there are sample code-related group keys configured in Unilab.

This problem is solved by the introduction of a new table (**UTSCGK**), in which all of the information from the individual group-key tables is merged. For this to work properly, two extra columns are added to the new table: one with the group-key name, the other with a sequence number, as a group-key name can be given several values for one and the same sample. The DB-APIs guarantee that this duplicated information remains consistent at all times.

4.2 Group-Key Definitions

It must be easy/simple for the application manager to create, remove and/or modify group-key structures at any time after the installation of the standard system. It is assumed that the application manager does not have detailed DBA knowledge (although knowing the basics about the meaning of a table in a RDBMS is required/assumed).

4.2.1 Group-Key Properties

Three categories of properties must be set for a group key:

- Properties related to the behaviour of the group key
- The group-key value list definition
- The functionality for group-key initialization and assignment.

4.2.2 Behaviour of Group Keys

Some group keys only allow one value for each object. This implies that the corresponding group key can be assigned only once to one and the same object. Some group keys are unique across objects. The group key **Sample code**, for instance, yields a list containing unique sample codes.

The **New value allowed** flag indicates whether or not the user is allowed to enter a group-key value that is not present in the specified list of valid values. When this flag is set, the user can dynamically extend the list of valid values. Note that the drawback of this absolute flexibility is that there is no longer any validation on the new values entered; this might compromise reporting!

The **Protected** flag indicates whether or not the user can enter a value manually or not. When this flag is set, it is no longer possible to enter a value for this group key through the client APIs. It is assumed that there is some customer-specific code either on the client or within the DB that will automatically assign a value for this key field.

For a group key, a default value can be specified. This default value is used when the group key is assigned to an object. This default can be overruled by the group-key assignment rule on object creation.

4.2.3 Value Lists for Group Keys

The value list for a group key can either be a fixed list or a dynamic list. Dynamic lists are defined through a SQL statement, fetching the values from a database table.

If the **New value allowed** flag is FALSE, then the dynamic list is used when the user types in a new value for the group key in the group-key property sheet.

4.2.4 Automatic Group-Key Initialization

The correct and automatic initialization of a group key can be defined by way of simple rules. Automatic rules greatly reduce typing errors. It also simplifies worklist construction/assignment (for routine/ day-to-day work).

Normally, the initialization of a group key only occurs once when an object is created! However, some assignment rules will also update the group key at a later stage (in the life cycle object).

Group-key assignment rules

The rules for group-key assignment are defined at the configuration level for each individual group key. The basis of the assignment rule depends on the object type for which the group key is defined.

Example: The group-key definition is a sample group key named **batch_nr**. The assignment rule for it is linked to what happens with an associated info field **Batch Number**. If, on the operational level, a sample is associated with an info field **Batch Number**, that sample will automatically be assigned a group key **batch_nr**. The Unilab system will see to it that the group-key value corresponds to the associated info-field value.

The events to look for (when automatically updating any group key) are extracted from the individual group-key definitions in a summary table (**UTEVRULES**). For the particular group key in the example, this table states that the assignment rules of the sample group key **batch_nr** must be verified each time the info field **Batch Number** is created, deleted or when its value is updated.

The occurrence of these and other critical events that require an action is monitored and processed by the (DB) Event Manager.

The info field-based assignment rule is only one of the standard supported assignment rules. Other standard assignment rules include the assignment of a sub-string extracted from sample type, sample code, request type, and request code. Finally, there is also option **Assignment rule none**. Custom assignment rules provide additional Unilab flexibility.

Note

A maximum of 400 items can be contained in a fixed list of values. If more than 400 values are needed, we recommend creating a custom table and retrieving all values for the corresponding group key by means of an SQL statement (which permits retrieving an unlimited number of values).

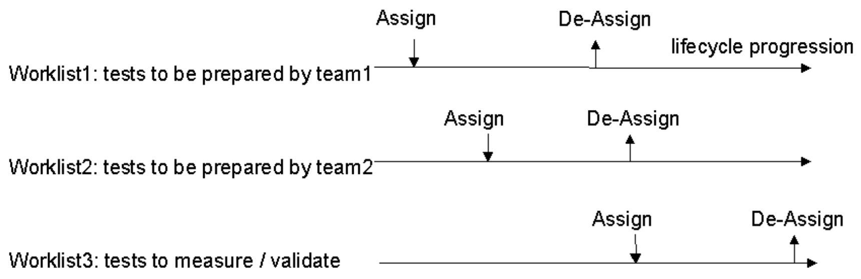
The width of a group key in a task bar can be defined. This is not done in the group-key definition itself, but in the task definition where the group key is used.

Worklist assignment rules

Worklist assignment rules provide yet another extension in the flexibility of group-key functionality. Indeed, based on the group-key assignment rules described above, a group-key table can only grow as new objects are created.

By means of worklists, one typically wants to view only the set of work to do. Hence these group keys must be removed (automatically) once the work is completed! In order to support such a worklist concept, instead of assignment upon object creation, assignment and de-assignment is made configurable at any specific moment in the lifecycle of the corresponding object. (e.g. de-assignment when the method status changes to **Completed** – e.g. the work has been performed).

The picture below shows it is possible to keep track of three different worklists with regard to method processing.


Assign
De-Assign
lifecycle progression
Worklist1: tests to be prepared by team1
Assign
De-Assign
Worklist2: tests to be prepared by team2
Assign
De-Assign
Worklist3: tests to measure / validate

Since wor*k*list assignment / de-assignment is triggered as a function of life-cycle progression, worklist assignment rules are independent of the group-key structure definition and are configured together with the life-cycle definition (= from within the **Define Life Cycle** application).

The following variables can be configured for each row in the worklist assignment window:

- Group-key name to which the worklist assignment rule refers (currently the only significant examples for this application have been identified for method group-key types; therefore, worklists are only supported for this group-key type).
- State of the method lifecycle in which assignment/ de-assignment should be triggered.
- An indication as to when assignment (IN) or de-assignment (OUT) is required in the given state.
- To determine the value to be assigned to the worklist for the given method, the use of standard method and sample properties is currently supported (e.g.. **method name, planned equipment, planned executor**). The possibility is also provided for custom function-based value assignment.

Since a method group key can be assigned both by group-key and worklist assignment rules, conflicts could arise. The group-key assignment rule is always evaluated first and can be overruled by the worklist assignment whenever the status of the method changes (note that the creation of the method object is also treated as a status change – from **nothing/not defined** to **<first status in the life cycle>**). In most cases, when a worklist assignment rule is used, it is customary not to specify any group-key assignment rule.

Note that it is also straightforward to move a method from one worklist to another as the method status changes. This could, for example, be exploited when one person is responsible for the preparation of the samples for a specific method while another person must actually perform the measurements.

Like for the group-key assignment rules, once again the Event Manager monitors the need for processing the worklist assignment rules. Indeed, whenever a status change occurs on the operational method, the Event Manager checks the worklist configuration assignment table and takes care of processing the relevant worklist assignment rules.

4.3 Inherit flag

Group keys can be inherited from the configuration level. This mechanism applies to samples (inheritance from the sample type) and requests (inheritance from the request type).

The Inherit flag can be set on two levels:

- On object level (e.g. sample type)
- On group-key level (e.g. sample group key).

Note

Note that, to permit inheritance from the configuration level (sample type) by the operational level (sample), both the corresponding sample-type and sample group keys must exist.

On the configuration level, the **Inherit group keys** flag can be specified. This can be done on sample type level (inherit group key by samples) and request type level (inherit group key by requests).

The definition of a group key includes a flag to specify whether or not this specific group key must be inherited from configuration to operational objects. Setting this property implies that the group key is always inherited, regardless of the configuration settings on the object level.

The **Inherit group keys** flag on the object level is always evaluated first. This guarantees that the basic rules are still valid. However, when the current **Inherit group keys** flag on the object level returns **No**, the corresponding setting on the group-key level is evaluated. If this flag returns **Yes**, the group keys are inherited.

Example: For group keys assigned to a sample type, the following combinations can occur for the **Inherit** flag:

SCGK	0	1	0	1
ST	1	1	0	0
Inherited ?	1	1	0	1

4.4 Use of Group Keys in the Task Bar

One of the primary goals of group keys is to restrict the object lists to a meaningful small set. When a group-key value is entered in the group key, the best matching group-key values are displayed.

In Unilab, two filtering elements based on group keys are provided:

- the Group-Key Pane;

- the Group-Key Bar.

For both filtering elements, it is not possible/allowed to use wildcards for any group key (it is either filled out or blank - in which case the selection key will be completely ignored), because this would drastically degrade the performance of the resulting selection query.

The Group-Key Pane

This filtering element is used to display lists in windows (e.g. the Sample List window, Request List window, etc.) and permits setting new group keys to filter the search on object lists in a more stringent manner.

In particular, the Group-Key Pane uses Boolean and comparison operators to create combinations of group keys that satisfy the user's needs. After having selected all the group-key elements to be used for the new search, the user must perform a Refresh: at this point, the object list will be populated with those elements that satisfy the set criteria.

To optimize the space on your application's desktop, the Group-Key Pane has been designed as pinnable. If you move the mouse over the pin button provided in the pane, the Group-Key Pane will slide out and become visible. If you want your Group-Key Pane to remain visible after it has slide out, just press the pin button. If you press the pin button again, the Group-Key Pane will become hidden once again..

The Group-Key Bar

Available in contexts in which an item must be assigned via a dialog box. Its position is fixed inside the dialog box. It permits setting a filter in the same manner as the Group-Key Pane.

4.4.1 The role of group keys in maintaining context when switching from one Task to another

Group keys are not only important in determining what will be actually displayed (i.e. the results of a query on the database) in a particular environment, but also the context that will be adopted when switching from one Task to another on the outlook bar.

The last item selected in one Task will be stored in memory upon switching to another Task and will be used as the initial context therein.

Keep in mind that context will be maintained from one Task to another, provided that:

- the values set on the Group-Key Pane in the first Task not be incompatible with those of the Group-Key Pane in the second Task;
- the match between the last item selected in the first Task and the item on which the focus will be in the second Task be one-to-one (e.g. if the last item selected in the first Task is a Sample that is present in more than one Request, then it will not be used as the context if you switch to the Request Lists task, as it would not be possible to determine which Request should be highlighted).
- the contents of the two Tasks be similar or somehow interrelated: e.g, the Basic Task of the Create Studies outlook page has nothing in common with the Basic Task of the Request Lists outlook page; therefore, it is impossible for the

item selected in the first environment to be used as the context in the second environment.

5 User Management

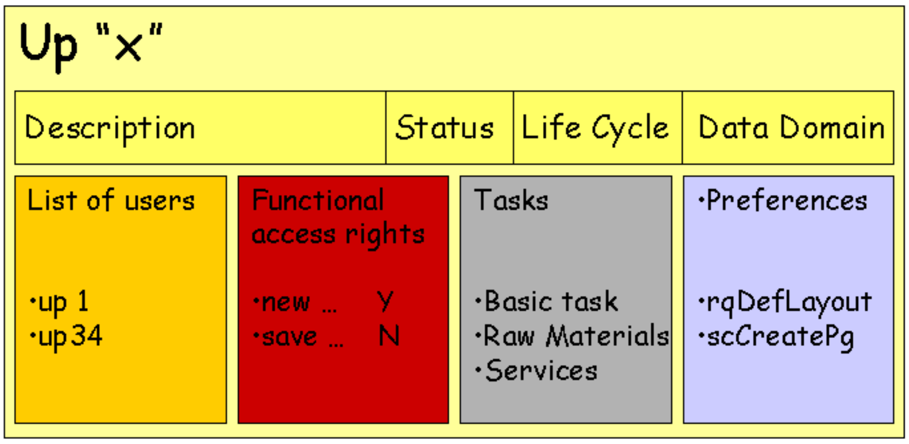
5.1 User Profiles and Users

Different type of users work with Unilab. Each user has specific data access rights and functional access rights. Moreover, for each Unilab user, the system must behave in a particular manner.

Users with the same data access rights and functional access rights are grouped into user profiles. For each user profile, the following properties can be defined:

- A description, life cycle and status in that life cycle
- Data access rights implemented by means of data domains
- Functional access rights
- Tasks
- Preferences

The picture below shows the properties of a user profile.



Each user can belong to several user profiles. In practice, a user switches from one data domain to another by changing his/her user profile.

5.2 Data Access Rights

5.2.1 Data Access Rights through Data Domains

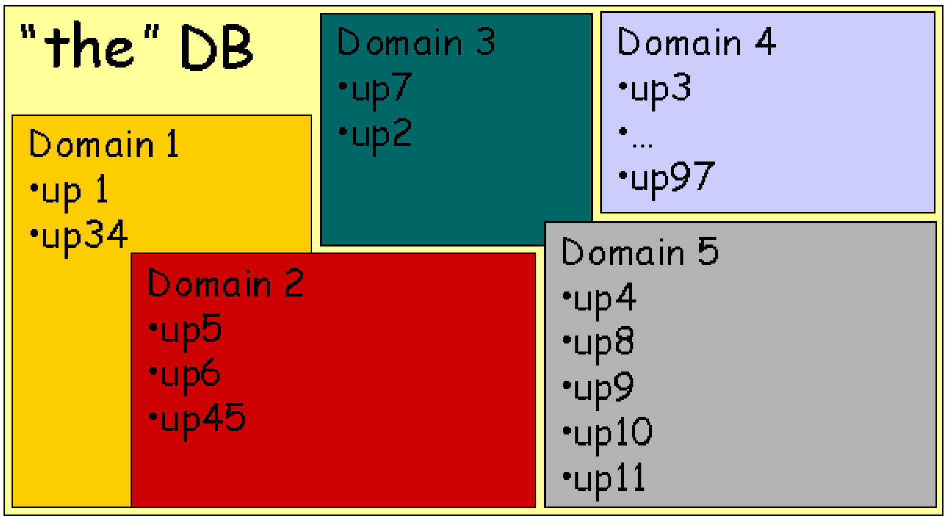
Data access rights are controlled through working areas called data domains. The default number of data domains is 16: however, at the time of installation, it is possible to extend their number to 32, 64 or 128. Each data domain holds the access rights to the individual objects.

The data access rights for user profiles are implemented by linking the user profiles to a data domain. Each user profile must be explicitly linked to one and only one data domain. The users belonging to that user profile access the data through the views of the corresponding data domain. Hence, they get the data access rights of that data domain. A user must have specific functional access rights in order to change the data domain of its default user profile. The LimsAdministrator is allowed to change the data domains of all user profiles. It is not possible to overrule the data domain for individual users.

To change the data domain, a non-LimsAdministrator user must add the user to another user profile that has access to the requested data domain.

One data domain can be linked to several user profiles.

The picture below shows how user profiles are linked to data domains.



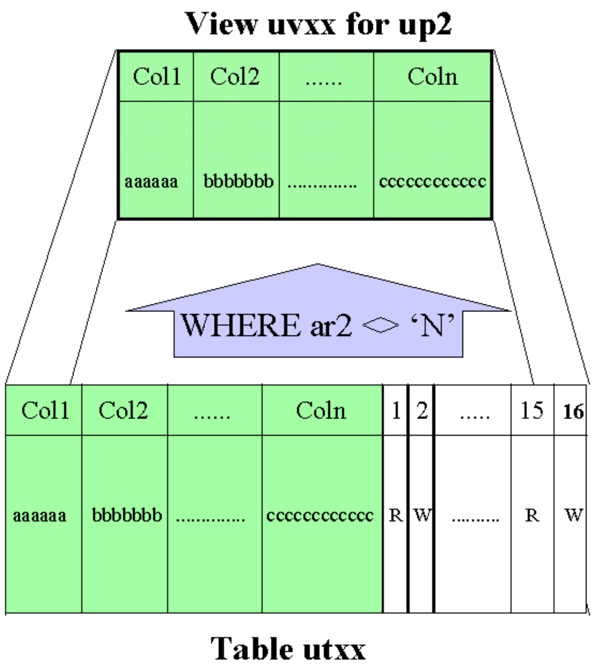
The data access rights for an object can be set to **Write**, **Read** or **None**. Note that the access rights for at least one data domain should always be set to **Write**. Otherwise, no one will be able to modify the properties of an object.

5.2.2 Implementing Data Access through Views

To prevent/restrict data access, the database **view** concept is associated with a particular data domain. Within the **view** concept, users belonging to a data domain do not see the content of a database table directly, but only after additional filtering has been performed by the database system.

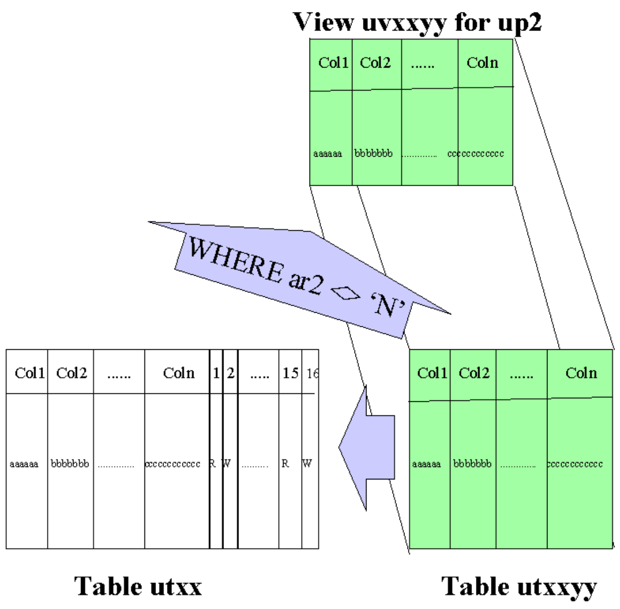
Space is provided in every row of each main object table to keep track of the access rights. In fact, each of these tables has been extended with 128 columns (AR1-128), each column providing space for an individual data access mark (W for read/write, R for read, N for none) for one data domain. The access rights columns themselves are not part of the view.

The picture below shows the access rights on main objects.



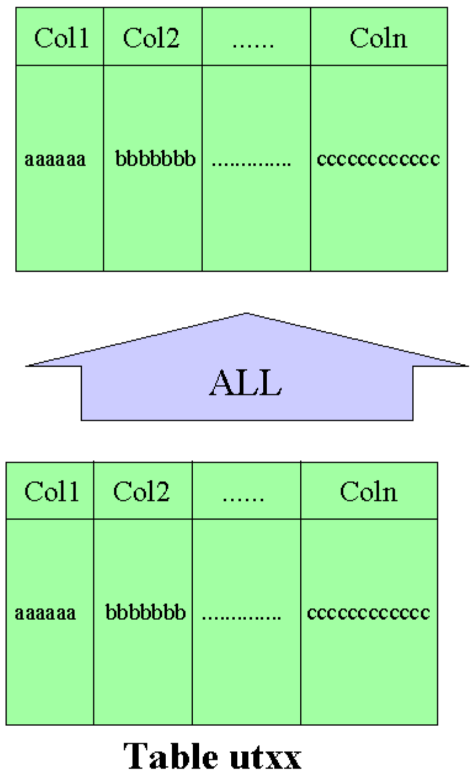
The access rights of used objects, such as parameter profiles assigned to a sample type, are exactly the same as for the main object itself (since these lists of used objects are considered to be an integral part of the – configuration – object).

The picture below shows the access rights for used objects.



Some configuration objects (life cycles, group keys, and attributes) do not have access rights. The views that are user-profile specific contain the same information as the corresponding tables. Access to these objects is primarily controlled through the functional access rights to the corresponding configuration application.

The picture below shows the view concept for objects without access rights.



5.2.3 Access Rights for the UNILAB DBA user

With respect to the UNILAB DBA user, Unilab makes some assumptions:

- The DBA has **up1** as the default user profile.
- **Up1** is assigned by default to dd1 (data domain 1).
- The DBA always has **Write** access to all objects**.**

It is therefore strongly advised that at least data domain 1 always have **Write** access to all objects.

The DBA is allowed to change the data domains of all user profiles.

Unilab does not automatically implement these assumptions. It is the responsibility of the system manager/application manager to see that these rules be met.

5.2.4 Accessing Configuration and Operational Objects

At the configuration level, access rights can be set for each individual object of any object type. These access rights are set in the main object table.

At the operational level, the access rights are applied in a strictly hierarchical way for **WRITE** and **NONE**. When the access rights for a sample are set to **NONE**, the users will not be able to modify the properties of the objects on lower hierarchical levels (info cards, info fields, parameter groups, parameters, etc.). Whether or not the corresponding methods are visible in the worklist depends entirely on the access rights of the method itself. If the method access rights are set to **WRITE** or **READ**, the method will appear in the worklist.

It is possible to modify an object (method cell, method, parameter,....) when one of its parents is part of a data-domain where the currently logged-on user has only Read-Only privileges if the system setting CASCADE_READONLY is set to NO. For instance, a method result can be entered for a parameter where the user has only Read-Only access to this parameter. Obviously, Read-Write access for the method itself remains mandatory.

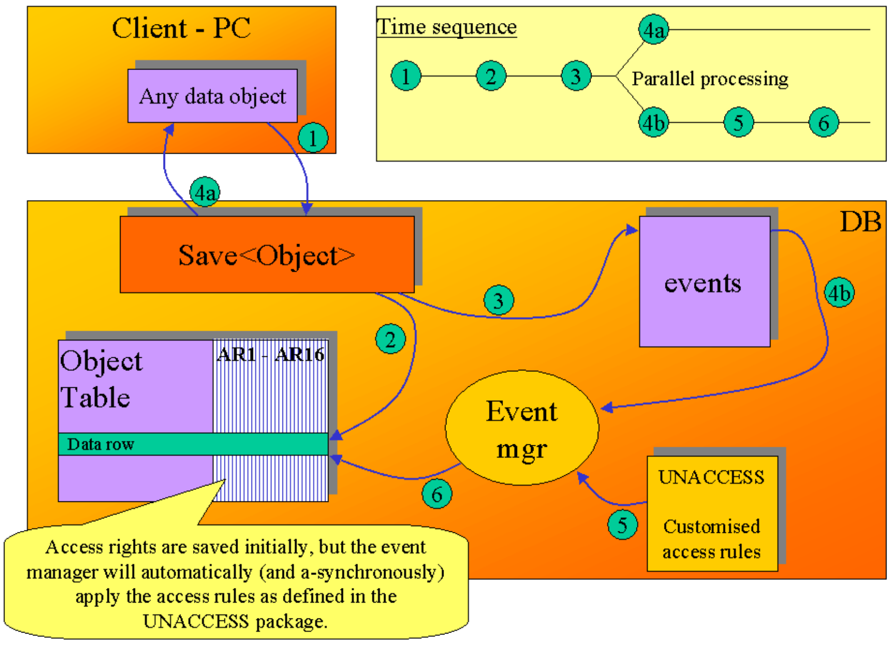
The propagation of this method’s result to the parameter is also allowed, provided that it is based on regular events handled automatically within the Unilab server.

Remark: If one of the parents has 'No Access' rights to this data domain, it remains impossible to modify the object.

5.2.5 Initialization of Access Rights

Whenever an object is created in the DB, the Event Manager a-synchronously initializes the access rights (by calling the **InitObjectAccessRights** function). The rules according to which the access rights are initialized can/must be customized to meet the project-specific requirements.

The picture below shows the initialization of access rights.



This mechanism also applies to access-right initialization of the operational objects. By default, objects on the operational level do not inherit their access rights from the corresponding configuration object. The access rights set on the configuration level strictly apply to the configuration objects themselves.

However, inheritance of access rights from the corresponding configuration object could be considered as an initialization strategy. To simplify this set-up (which is not the default approach), the InheritObjectAccessRights() has been provided; this function can/must be called from within the InitObjectAccessRights().

5.2.6 Update of Access Rights

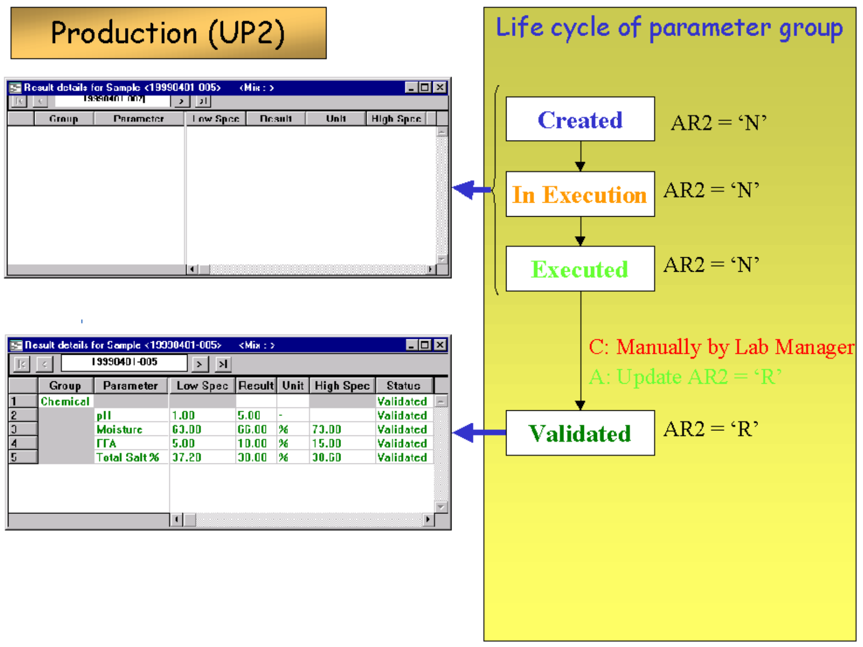
The access rights of any object can be updated manually at any time (on the object's property sheet).

Access rights are checked only when certain key operations in the object's lifecycle are performed. In most cases, this coincides with the object's creation. However, it is possible that Write access is granted to a particular data domain during a specific period within the full lifecycle of the object. The access rights can be updated by an action assigned to the relevant state transition. The access rights of an object can also vary/change according to the status of the object. The update of access rights must be triggered from within an action by calling the UNACCESS.**UpdateObjectAccessRights()** function.

Example: In a production environment, samples are taken along the production line and sent to the laboratory. The returned results are used to adjust the subsequent steps in the production process. However, no decisions should be made on partial results. Therefore, analysis results should only become available for production after the lab manager has validated all results.

This is implemented by setting the initial/default access rights for the data domain of user profile **Production** to **None**. Parameter access rights are not updated to Read until after the results are validated.

The picture below shows how the results become visible to production after validation.



	Group	Parameter	Low Spec	Result	Unit	High Spec	Status
1	Chemical						Validated
2		pH	1.00	5.00	-		Validated
3		Moisture	63.00	66.00	%	73.00	Validated
4		FFA	5.00	10.00	%	15.00	Validated
5		Total Salt %	37.20	30.00	%	30.60	Validated

5.3 Functional Access Rights

Independent from and complementary to the data access rules, functional access rights are needed. It is, for example, possible that a specific user is only allowed to fill in results, but is not allowed to create new samples or run the configuration application.

The functional access rights determine whether users of a certain user profile have the right to start up a specific application, or on a more detailed level, which functions within this application are accessible or inaccessible for these users.

Functional access rights (as well as tasks) can be seen as an additional security mechanism that directly reflects on data access. In fact, data can only be accessed/manipulated by the user through a specific application. Such an application typically provides an additional filter on what the user can actually do with the data he/she can access. If a user cannot access the functionality to create new samples in the operational modules, he/she cannot create samples, even though data access rights might suggest that he/she be allowed to do so.

To simplify maintenance, functional access rights can be specified on the user-profile level; these settings are automatically applicable for each individual user within that profile. However, to allow sufficient flexibility, it is possible to specify exceptions for each individual user!

5.4 Tasks and Preferences

The way an application behaves is obviously influenced by the data access rights, as well as by the functional access rights that have been defined for a given user profile. Two additional mechanisms can be configured to control the behaviour of the applications: tasks and preferences.

5.4.1 Tasks

A task consists of a specific group-key combination that allows a user to quickly access the data required to execute a certain laboratory task.

For each user profile, a list of tasks to be performed by the users of that profile can be defined.

5.4.2 Preferences

The presentation details of the data are largely controlled by preference settings. These preferences define the contents and/or behaviour of individual windows.

Tip

A detailed description of the different preferences is available in the On Line Help of the **User Management** application.

Types of preferences

Although the list of preferences is vast and diverse, some conventions have been applied.

Preferences for system behaviour

In general, preferences defining a certain behaviour have a name with the following format:

<xx><Behaviour>

where:

- <xx> refers to the object for which the preference is set
- <behaviour> refers to the Unilab behaviour or property that is set.

Examples:

- **ScCreatePg** controls the creation of parameter groups (<behaviour> = **CreatePg**) for newly-created samples (<xx> = **Sc**)
- **PaResultEditable** controls whether the parameter (<xx> = **Pa)** result is editable (<behaviour> = **ResultEditable)**
- **ScInheritActualStGk** controls if the current sample-type group-key values used during sample creation are inherited (<behaviour> = **InheritActualStGk**) by the newly-created samples (<xx> = **Sc**).

Preferences for setting of default layout

Many preferences are involved in setting the default layout for individual windows. The names of these preferences have the following format:

<xx>Def<window>Layout

where:

- <<xx> refers to the main object in the application
- <window> refers to the window for which the layout is set (optional).

Example:

- **ScDefCreatelayout** sets the default layout for the **Create sample** window (<window> = **Create**) in the sample list (<xx> = **Sc**)
- **RqDefCreatelayout** sets the default layout for the **Create request** window (<window> = **Create**) in a request list (<xx> = **Rq**)
- **ScDefPaLayout** sets the default layout for the **Parameter results** window (<window> = **Pa**) in a sample list (<xx> = **Sc**).

Note that the <window> part is not included in the preference name to specify the layout of the main object list in the application.

Example:

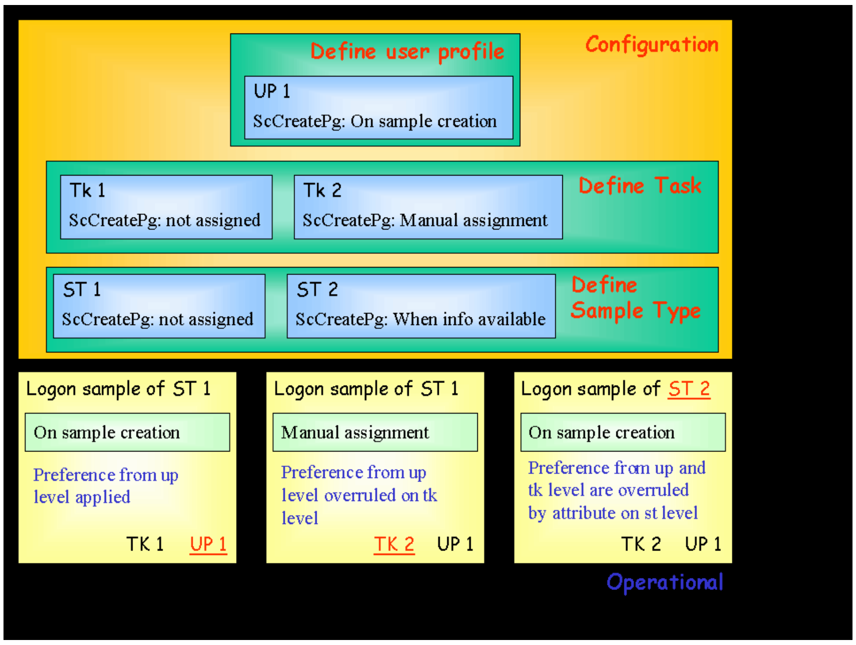
- **ScDefLayout** sets the default layout for the sample list (<xx> = **Sc**)
- **StDefLayout** sets the default layout for the sample type list in the **Configuration** application (<xx> = **St**).

Preferences for setting of default task

Specific tasks can be configured for the creation of new samples and requests. These tasks can be set as default tasks for the corresponding windows using preferences like **<xx>DefCreateTask** (with <xx> = **Sc** or **Rq**).

Scope of preferences

The picture below shows an example (for **ScCreatePg**) to illustrate the concept of overruling preferences on different levels.

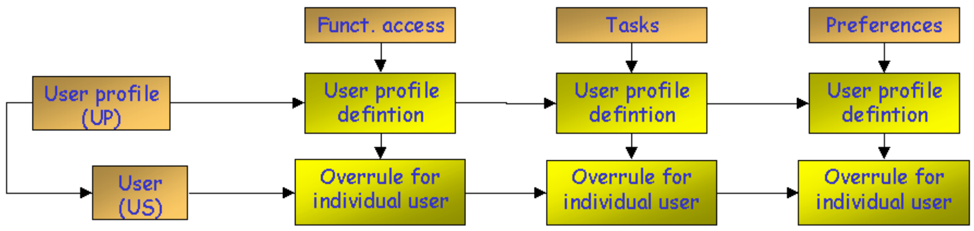


The default values for each of these preferences are set system-wide in the **utpref** table. These default settings are inherited at the user-profile level. The default preference values can be overruled on the user-profile level. The settings on the user-profile level can in turn be overruled for individual users. A number of preferences can again be overruled on the task level. Preferences set on the task level always overrule the corresponding user-(profile) preferences. For instance, the task preference **meResultEditable** overrules the settings for the user-profile preferences **meResultEditable**.

5.5 Overruling User-Profile Defaults for Individual Users

It is possible for individual users to overrule one or several of the user-profile defaults on the configuration level.

The picture below shows the inheritance-based mechanism that overrules user-profile defaults for individual users.



Not only the functional access rights and tasks, but also their properties can be overruled on the user level. Thus it is possible to deny certain functionality to a specific user, while all other users in this user profile can perform this function.

For tasks, the **Hidden** flag and the default value of a task's key fields can be overruled per user. The structure of the task itself obviously cannot be changed, as this would result in a different task.

5.6 Unilab Users

5.6.1 Creation of Unilab Users

A Unilab user is initially created as an address in the Unilab database. The user can then be selected as a value for info fields or attributes referring to the address list. However, users that are created in this manner are not yet active Unilab users, as they cannot log on to any of the various Unilab applications. To become an active Unilab user, a corresponding Oracle user must be created.

5.6.2 Unilab Users versus Oracle Users

To become an active Unilab user, an Oracle user corresponding to his/her address must be created. This is done by setting the **Unilab User** flag in the user properties. When this flag is set, all necessary Oracle structures are created (after explicit confirmation). In fact, an active Unilab user corresponds to an Oracle user. Likewise, the user is deleted as a DB user when the **Unilab User** flag is switched off for a certain user. However, all references and properties of that user are kept in the address book, but the user can no longer log on to the database.

When activating a user, a default user profile must be specified. Each application the user accesses will start up with this user profile. The user will then get all preferences, functional and data access rights as specified for that user profile (unless individual settings have been overruled on the user level). The application will also start up with the default task for that user profile. When a user belongs to several user profiles, he/she can always switch to another user profile within the application.

5.6.3 User Passwords

Case sensitivity of user names and passwords

When logging on to an application, a user must identify him/herself by entering his/her user ID and password. When an application is started, it first connects to the Oracle database. Then a number of default settings, such as the default task and corresponding preferences are retrieved. The user name is not only used to connect to the database, but also to verify whether the user and default user profile are both active.

Oracle usernames are case-sensitive. In practice, this implies that the User ID **ALAN** and **Alan** are not identical on the Oracle DB level (and do not have the same privileges). Therefore, these two users are considered to be two different users from a Unilab point of view.

Important It is strongly recommended that user IDs not use case sensitivity to make a distinction between users.

The password is used exclusively to connect to the database and is case-insensitive.

Temporary initial password

When a new user is created, the initial password is sent by email to both the user and the DBA. During connection for the first time using the initial temporary password, the user will be forced to change this password. A user can alter his/her password from within any Unilab application, at any time. For security reasons, the modification of a password is only accepted if the previous password is entered correctly.

Password expiration and security

In addition, the application manager can specify the duration of password validity. That is, he/she can set rules to determine when a password expires and, consequently, must be modified by the user. A warning is automatically displayed by the system when the user’s password has expired. The application manager can also specify masks for the password. Users can thus be forced to choose non-trivial passwords. The implementation of password expiration and security is based on Oracle.

5.6.4 Experience Levels and Qualifications

Not all users are allowed/authorized to perform any method. Proper training and education is required in order to be able to execute a specific method adequately. Authorization to execute a method can be specified on the method level:

- The experience level that is required to execute the method
- A specific list of users that are authorized to execute the method
- Or a combination of these two options.

It is necessary to specify the acquired experience level(s) for the individual users. A user that has acquired a specific experience level can execute each method for which that experience level is required (or for which he/she is specifically assigned to be an authorized executor). The picklist of experience levels can be defined/customized according to enterprise needs. This can be achieved by editing the **utel** table.

In many laboratories, additional fine-tuning is required. For example, it is possible that a user who is being trained to acquire a specific experience level is not yet allowed to execute a method. Therefore, the qualification can be specified for each experience level of a certain user. The qualifications themselves can be edited in the **utqualification** table. Which qualifications are required in order to effectively execute a method can be set system-wide in the **EL_QUALIFICATIONS** system setting.

Example:

With the following settings for the qualifications:

Qualification	Description
"0"	"Not qualified"
"1"	"In training"
"2"	"Beginner"
"3"	"Fully qualified"

And the required experience qualification (in **utsystem**) set to:

SETTING_NAME	SETTING_VALUE
EL_QUALIFICATIONS	**"2 3"**

The system will only allow users with the qualification **Beginner** and **Fully qualified** to execute a method. A user with any other qualification will not be able to execute the method, even when they have the correct experience level assigned!

5.6.5 Additional information

Additional information on Unilab users can be found in Technical Note “SIMATIC IT Unilab – Security – Database and Logon” in the Unilab Product Library.

6 Address Book

In Unilab, it is possible to maintain an address book containing details about all people involved with the lab. The system recognizes different types of addresses (and extra types can easily be added):

- **Unilab users:** These are the actual Unilab users, who belong to at least one user profile. Any of these users will have access to the system, as defined by the functional access rights of their (default) user profile;
- **Suppliers:** These are primarily the company's suppliers, who supply instruments or other materials to the lab or provide the factory with raw materials;
- **Customers:** These are the company's customers. They may be customers of the lab (e.g. in service labs) for whom analysis requests are handled, or customers of the factory to whom finished products are shipped.

Properties of an address

Each address is identified by a unique address ID. This ID cannot be modified after it has been saved to the database. Due to the fact that active Unilab users correspond to Oracle users, the Oracle user ID naming restrictions apply to the address ID.

The following naming conventions apply:

- Names must be from 1 to 20 characters long.
- Names are case-sensitive.
- Passwords are case-sensitive.
- Names can only contain alphanumeric characters **a** through **z** and the character _
- A name cannot be an Oracle reserved word.

Tip

For a detailed description regarding Oracle naming conventions, refer to the Oracle documentation.

7 Tasks

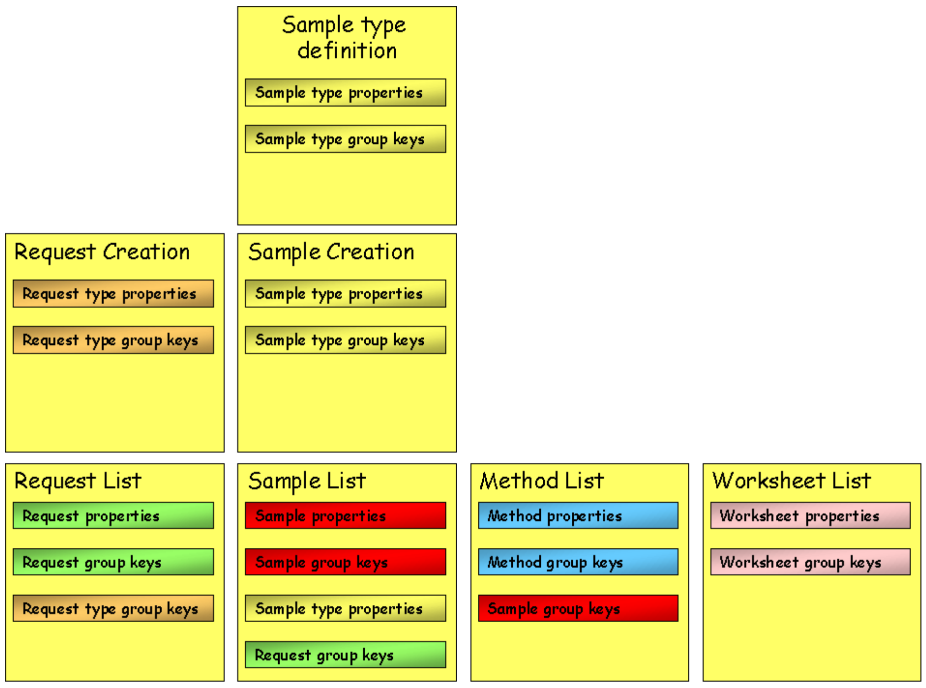
7.1 Task Definition

A task consists of a specific group-key combination that a user needs for quickly accessing the data required to execute a certain laboratory task.

Tasks allow focusing on a subset of data. Because fields can be hidden (within a task definition) and filled with a default value, a task definition (based on the data access rights) can also provide a powerful additional filter for differentiating what data can be accessed by specific users. One could e.g. imagine servicing several labs from a single database. Given the fact that, on a conceptual level, analysts typically are granted a specific set of privileges, these could be bundled into the same user profile (with a uniform set of 'generic' analyst-related data and functional access rights). If, however, one does not want to grant analysts access to one another's data, this could be implemented on the 'task' level. Indeed, the actual data to which a specific analyst has access can be limited by inserting a hidden group key **Lab** with a user-dependent default setting (i.e. the name of the analyst's lab) into each of the analyst's task bars.

7.2 Defining Tasks

For task definition, not only group keys related to the main object, but also any standard property of the main object can be selected as a key field of the task. It is also possible to use group keys from a higher hierarchical level. The picture below shows an overview of all different lists for which tasks can be defined, as well as all possible field combinations for these tasks.



Task properties

For each field used in a task, a number of properties can be set, such as:

- The width of the field in the task bar
- The default value of the field (fixed or dynamic)
- The sorting options for the field-value list
- Flags to control field behaviour, such as the **Hidden** flag, the **Protected** flag, the **Mandatory** flag and the **Auto refresh** flag

Task-specific preferences

Task definition can be extended with a set of preferences. If configured, these preferences always overrule the corresponding preferences set on the user (profile) level. For example, when the **<xx>DefLayout** preference is set on the task level, this permits linking a particular window layout to a task.

Tip

Only a limited set of user-profile preferences can be overruled on the task level. Refer to the On Line Help of the Define Task application for the complete list.

7.3 Defaults in Tasks

Each task field can have a default value. These default values can be fixed or dynamic. The default values are used in tasks to implement filters on the data so that users only view/modify the data they are interested in or are allowed to view/modify.

In case of fixed default values, the value must be defined explicitly for each user. This is acceptable with a limited number of users. It is not the case on systems where users are created on a daily or a weekly basis. Dynamic defaults allow specifying the default values in a more dynamic manner by means of tilde substitution.

The table below lists the default values that are supported.

Kind of default value	**Description**
Fixed value	Fixed
~us@<std_prop>~	Is replaced by the corresponding standard property for the connected user (used view=**uvad**).
~usau@<au>~	Is replaced by the corresponding user-defined attribute for the connected user (used view=**uvadau**).
~up@<std_prop>~	Is replaced by the corresponding standard property for the current user profile of the connected user (used view=**uvup**).
~upau@<au>~	Is replaced by the corresponding user-defined attribute for the current user profile of the connected user (used view=**uvupau**).
~=Function1~	Is replaced by the result of the called function.

Note that, although a user is configured by means of an address (**ad**), the more user-friendly notation **us** is used for tilde substitution in tasks.

7.4 Flags for Task Field Behaviour

Hidden flag

Switching on the corresponding flag can hide fields in a task bar. For hidden fields, a default value must be set in the task definition. Hiding fields (based on the data access rights) is an additional filter for differentiating what data can be accessed by specific users .

Protected flag

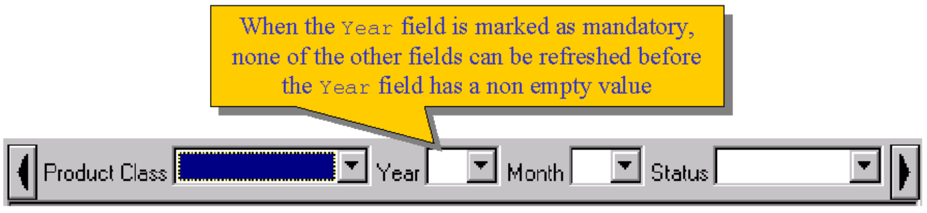
Fields for which the protected flag has been switched on can no longer be edited/modified by the user. A default value can be set in the task definition.

Unlike hidden fields, protected fields remain visible in the taskbar. Hence, the user can consult the default value set in the field. Such a field does not appear as a normal dropdown list box, but as a protected edit box.

7.5 Mandatory Task Fields

Mandatory fields must contain a value before any other field list can be refreshed/loaded.

The picture below shows the mandatory field in the task bar.


When the Year field is marked as mandatory, none of the other fields can be refreshed before the Year field has a non empty value
Product Class
Year
Month
Status

When multiple fields are set as mandatory, it is not possible to refresh any of the other fields (non-mandatory fields) until all mandatory fields contain a value.

7.6 Customizable SQL in Group Key bar

The SQL statements for group keys can be customized in the **Define Tasks** application.

When you open a task, the query for the different components can be customized by selecting the SQL value in the Value List Tp drop-down list, and specifying the SQL statement in the SQL column that became editable.

Please note that such customization requires expert database knowledge. Poor configuration may cause Unilab to behave incorrectly.

7.7 Task-Bar Performance

Task-bar performance is determined by the performance of the queries linked to the task. Two types of queries are executed in a task.

The first type of query is executed upon initialization of the lists of values for the individual group keys or standard attributes used in the task. These lists of values are established upon initialization of the default task during application start-up or when the task context is switched in an already-open application. The executed queries select all existing and distinct values for the fields used in the task.

The second type of query is performed to fill the window lists (sample list, method list, etc.). All values thus obtained apply to the current group-key selection.

The performance of the aforementioned queries is mainly controlled at the Oracle level. In addition to the Oracle rules, some concepts have been introduced into Unilab to enhance overall task-bar performance.

Initializing individual field lists

Fetching the list of distinct values for one individual field can take a lot of time, given the number of records that must be scanned. On the other hand, many lists of values are rather static. The list of sample states, for instance, does not change that often. Therefore, it does not have to be refreshed each time the task is initialized. Hence, only refreshing those lists of values that change dynamically enhances the performance of task-bar initialization significantly.

This behaviour can be obtained by setting the **Auto refresh** flag for individual fields. This flag allows controlling when the field list must be generated. For static lists, the value can be set to **Only once**. For dynamic lists that change frequently and, hence, need to be refreshed each time the task is initialized, this flag can be set to **When loaded**.

Initializing all fields in a task bar

When a task is initialized, all the value lists for all fields must be initialized.

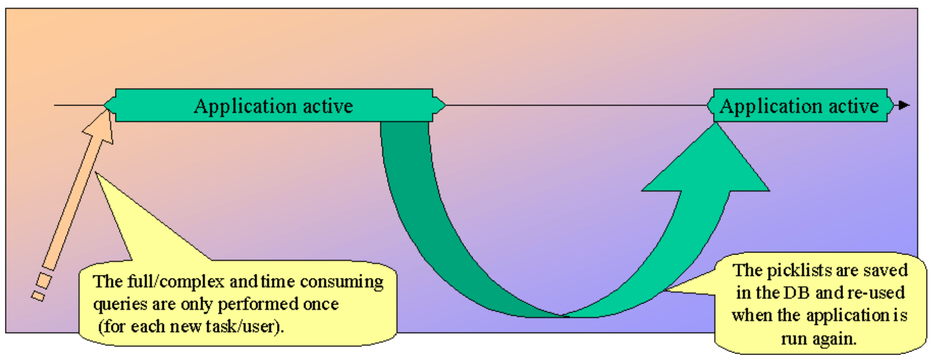
Task-bar initialization performance can be enhanced by introducing additional keys in such a way that the number of searched records is drastically reduced. When the task’s context is switched, the value lists of task fields are automatically restricted to the value selected in previous fields.

To achieve this, the order of the key is taken into account: the leftmost key in the task bar is considered to be the most restrictive. To prevent users from modifying these key fields, the corresponding fields can be set as hidden or protected. As a general rule, one can say that the task bar should start with the most discriminating group key and finish with the least discriminating key.

Task-bar performance upon application start-up

Application start-up time is drastically reduced thanks to a mechanism that saves the previously-used values of the taskbar for future reutilization. When an application is started up again, the user will get exactly the same list of values that they had before the application was stopped.

The picture below shows group-key performance in the taskbar.


Application active
Application active
The full/complex and time consuming queries are only performed once (for each new task/user).
The picklists are saved in the DB and re-used when the application is run again.

The user can decide at any time to refresh and save the currently-used group-key value lists. This solution/approach was chosen because it was noticed that a lot of the group-key value lists are more or less static (= do not change that often). Thanks to this mechanism, the user has full control over when which list of values will be refreshed.

Oracle rules and task-bar performance

Retrieving the list of distinct values for one field can still take a lot of time, given the number of records that must be scanned. Performance can be influenced by considering a number of parameters on the Oracle or database level, such as:

- The combination of group keys and standard properties in a task (using standard properties has a negative influence on performance);
- Whether or not standard properties are indexed;
- Mixing group keys of different kinds should be tested carefully;
- The uniqueness of a group-key value and how it is related with the frequency of use of this value;
- The Boolean operator OR;
- The comparison operators that differ from "=" (LIKE, <, >, ...);
- The special value <NULL> (which is translated into IS NULL or IS NOT NULL in the where clause).

8 Layouts

The layout of a data table can be configured to visualize the prominent properties of the listed data.

A layout is an ordered list of columns with their characteristics. The characteristics of a layout column consist of:

- Visual characteristics: alignment, title, width, etc
- Edit characteristics: protected or editable
- Data characteristics: data source, data type, etc.

8.1 Defining Layouts

The definition of data-table layouts is edited in the **Define Layout** application. For each specific layout, there are a number of rules. These rules are described in the on line help of the **Define Layout** application.

Tip

Refer to the **Online Help** of the **Define Layout** application for the properties of specific layouts.

8.2 Using Layouts

The default layout for almost any table object in the system can be set per user (profile) or per task by means of specific preferences.

Note

User-profile preferences can be overruled for a specific user. If layout preferences are set on the task level, they always overrule the corresponding layout preferences on the user (profile) level.

When multiple layouts are defined for a specific list, the user can switch between the layouts in the application itself. Note that performance might be affected when changing the layout of an object list to a layout displaying more details (no refresh of data). In this case, it is recommended that the following procedure be adopted to change the layout:

- First minimize the object list
- Then change the layout
- Finally, open the object list again. The new layout will be applied.

Performance is not affected when changing the layout for an existing sample list to a layout containing less detail.

8.3 Layout Editing from within the Data Table

Authorized users can edit the data-table layout in the application itself. The following layout features can be modified from within the data table:

- The sorting order used in the layout
- The order of the columns in the layout

Users with the appropriate authorization can save these modifications into the database. From that moment, the changes made will apply for all users.

8.4 Sorting

The sorting order of the data displayed in a data table can be set on different levels:

- On database side, the application manager can set the default sorting in a specific layout definition
- On client side, the user can sort the data in a data table by clicking in the corresponding column header. This sorting overrules the default sorting set in the layout definition.

Sorting in layout definition

In the layout definition, the application manager can define the sorting order and sorting option (none, ascending, descending) for each of the layout columns.

Sorting in a layout definition can only be done based upon the properties of the main object.

Sorting on the sample list, for instance, can be done based upon the sample properties. However, sorting of the method list (work list) can only be done on standard properties of the methods or method group keys. To provide sorting on the sample code, the sample component from the method properties (**me.sc**) should be added to the layout instead of the sample component from the sample properties (**sc.sc**).·This also results in better performance, since it could avoid an additional DB API call (all the sample properties should not be loaded unnecessarily).

The items used for sorting must be part of the layout, but they need not necessarily be visible. If required, they can be hidden.

Sorting on client side

Sorting that is executed by clicking on the column header in a data table is a pure client process. After refresh of the data in the data table, the default sorting order specified on the database level is re-applied.

The application manager can also prevent client sorting on specific columns by setting the **Order** property of the corresponding column to the value **-1**. This is applied by default for layout columns on which client sorting leads to confusing situations, such as columns containing numeric values that are filled out by the end user. An example is the column to specify the number of samples of a sample type in the **New sample** window.

A negative sorting order (-1, -2, etc.) can be applied in case the columns should be sorted by default, but columns that are not sorted on the client side are allowed.

Sorting can be executed on the client side when the **Order** property of the column has value 0.

8.5 Tree-View Behaviour

A number of data tables assume a behaviour similar to that of a tree view. For example, this is the case for:

- The **Sample type** list in the configuration application: for each sample type, the list of info profiles (**stip**) and the list of parameter profiles (**stpp**) are displayed
- The **Parameter results window**
- The **Assign Full Test Plan** window.

The hierarchical view in these data tables is obtained through a combination of multiple layouts. In the parameter results window, for instance, three different layouts are involved:

- **Parameter group results** layout
- **Parameter results** layout
- **Method results** layout.

The actual tree view behaviour itself is controlled through a number of settings. These settings include:

- Multiband Style (vertical or horizontal)
- Column sizing style (width of columns on different levels synchronized or not)
- Column headers on the different bands visible or not and text background colors for each band to enhance visibility.

These settings are not part of the actual layout definition itself. Since these settings are not stored in the database, the actual tree view behaviour can be different for each client PC, or even for each user on one client PC.

9 Unique Code Masks

9.1 Definition

For several objects on the operational level (e.g. samples, requests, worksheets), unique identification codes are required. The code-generation algorithm is controlled by way of unique code masks (UCM).

Unique code masks can also be used for other numbering sequences within Unilab, such as run numbers or certificate numbers.

9.2 Structure of Code Masks

A unique code mask can be defined as a structure consisting of a number of components. The following components can be used:

- Any fixed text (mainly used as separator to enhance readability)
- Date and time fields
- A number of special fields
- Counters (= sequences)
- Special fields and counters for Centralized Quality Management

Date and time fields

Date and time fields can be used to insert the current value of any part of the time stamp into the code mask.

The table below lists some examples of available date and time fields in unique code masks.

Field	Description
Y	last digit of year
YY	last 2 digits of year
YYYY	year
IY	last digit of ISO year
IYY	last 2 digits of ISO year
IYYY	ISO year
MM	month 01...12
MMM	month JAN...DEC
WW	week number
DDD	day of year
DD	day of month
D	day of week
hh	hour 00...23

Field	**Description**
mm	minutes
Ss	seconds

Note

When working in a multiple time zone environment, the structure of used code masks must be build up carefully in order to avoid possible problems. For more information about unique code mask behavior in a multiple time zone environment, refer to the Multiple Time Zone documentation, in the Unilab Product Library.

Special fields

A number of special fields can be used in code masks. Some special fields allow inserting properties of an object into the code mask. Other special fields allow including part of the object creation context in the unique code (for instance, user ID of user who logs on a sample).

For each of these special fields, the application manager can choose to use the entire value or only a substring.

The table below lists the available fields you can use in unique code masks.

Field	**Description**	**Use**
st	The sample type of the sample for which the unique code mask is to be used	Samples only
au	The value of the attribute which is specified (and associated to the sample type of the object for which the unique code mask will be used)	Samples only
userid	The current user-id	• Samples • Requests
rt	The request type of the request for which the unique code mask is to be used	Requests only
rtau	The value of the attribute which is specified (and associated to the request type)	Requests only
rq seq0	Unique sequence number for samples within the same request. Is reset to 0 for each new request	Samples that belong only to a request
rq seq1	Unique sequence for samples belonging to the same request. Is reset to 1 for each new request	Samples that belong only to a request
rq	The request to which the sample for which the unique code mask is to be used is assigned.	Samples that belong only to a request
upau	User-profile attributes	• Samples • Requests
adau	User attributes	• Samples • Requests
wt	Worksheet type of the sample	• Samples • Worksheet
pt	The protocol of the study for which the unique code mask is to be used	Protocols only

ptau	The value of the attribute which is specified (and associated to the protocol of the study for which the unique code mask will be used).	Protocols only
sd seq0	Unique sequence number for samples within the same study. Is reset to 0 for each new study.	Samples that belong only to a study
sd seq1	Unique sequence for samples within the same study. Is reset to 1 for each new study.	Samples that belong only to a study
sd	The study for which the unique code mask is to be used.	Studies only

Counters

Counters are structures that are used to generate unique sequence numbers. A counter has a value range (i.e. its minimum and maximum values), as well as an increment specified.

A counter can be reset before it reaches its maximum value. The moment when the counter must be reset can be specified (as a qualifier). By default, the counter is reset on a change of the preceding field in the mask. If no preceding field is found, then the successive field is used.

Example: For the code mask **{YY}{MM}{DD}-{My_counter}**, the counter will be reset on change of DD, since "day" is the preceding field in the mask.

There are four predefined counters:

- **DayCounter**
- **WeekCounter**
- **MonthCounter**
- **YearCounter**

that can be used in combination with sample/request planning (ahead of time). These counters keep track of the number of calls already assigned within the specified period of time.

The table below gives an example of behaviour for some of these counters. All samples are created on the same day, but, for some of them, the sampling date is different.

Structure	**Generated codes**
{YY}{MM}{DD}-{DayCounter\3}	• 951117-001 • 951117-002 • 9511**18**-001 • 9511**17-003**
{YY}{MM}{DD}-{MyCounter\DD}	• 951117-001 • 951117-002 • 9511**18**-001 • 9511**17-001**. **Note**: This code mask will fail since this specific sample code already exists in the system.
LAB{userid\left(2)}{DDD}-{MyCounter\DDD}	• LABpb234-001 • LABpb234-002

	• LAB**si**234-003 • LABpb**235**-**001**

Special fields and counters for Centralized Quality Management

Special fields and counters are available when defining a unique code mask in a Centralized Quality Management context.

The table below lists the fields that can be used in unique code masks for Centralized Quality Management.

Field	**Description**	**Use**
<<pp_key[1..5]>>	The parameter profile key of the object for which the unique code mask is to be used	sc, rq, sd, ws
gk.<<pp_key[1..5]>> seq0	Unique sequence number for the object within the same context, starting from 0	sc, rq, sd, ws
gk.<<pp_key[1..5]>> seq1	Unique sequence number for the object within the same context, starting from 1	sc, rq, sd, ws

<<pp_key[1..5]>> must be replaced by the effective id defined in the table utkeypp.

Example:

Unique codes generated for the following unique code mask structure:
{plant\Left(3)}-{YY}{MM}{DD}-{gk.plant seq1\3}

Sample	**Plant**	**Sample code**
sc1	London	Lon-060403-001
sc2	London	Lon-060403-002
sc3	Brussels	**Bru**-060403-**001**
sc4	Brussels	Bru-060403-002
sc5	London	Lon-060403-003
sc6	Brussels	Bru-**060404**-**003**

A counter for Centralized Quality Management can only be dependent of 1 context value. The current value of such counters can be checked in the table **utucobjectcounter**.

Example:

Unique codes generated for the following unique code mask structure:
{plant\Left(3)}-{st\Left(3))}-{YY}{MM}{DD}-{gk.plant seq1\3}

Sample	**Sample type**	**Plant**	**Sample code**
sc1	Milk	London	Lon-Mil-060403-001
sc2	Yoghurt	London	Lon-**Yog**-060403-**002**
sc3	Milk	Brussels	**Bru**-**Mil**-060403-**001**
sc4	Yoghurt	Brussels	Bru-**Yog**-060403-**002**
sc5	Milk	London	Lon-Mil-060403-003
sc6	Yoghurt	Brussels	Bru-**Yog**-060404-**003**

9.3 Configuring Unique Code Masks

Unique code masks are defined in the **Define unique code mask** application. There is no restriction to the number of unique code masks that can be configured. Unique code masks should be defined with great care, because the code they generate is the primary key to uniquely identify a number of objects within the Unilab database.

The system does not perform a test on the correctness of a mask for a particular use. It is quite possible that two code masks, with different fields, eventually yield duplicates on the operational level. This is NOT checked up front! However, any **CreateSample** function using the non-unique sample code will fail. Such conflicts must be avoided!

Counter behaviour

Keep in mind that the counter is a single object that can be used in the structure of different code masks. In fact, using the same counter in two different code masks is allowed, although not recommended, as the counter increments sequentially, regardless of whether it is counting operational objects of the same type (e.g. all samples, requests), or whether the operational objects being counted belong to different types. This will lead to gaps in the sequence of codes generated for each type of operational object being created. The example illustrates the codes that are generated upon creation of 3 samples, followed by 2 requests, which, in turn, is followed by 2 samples, all created on the same day:

- Structure of Sample Unique Code Mask: YYYYMMDD-(My Counter/XX)
- Structure of Request Unique Code Mask: YYYYMMDD-(My Counter/XX)

Operational Object Created (on Sept. 20, 2006)	**Unique Code Generated**
Sample 1	20060920-**01**
Sample 2	20060920-02
Sample 3	20060920-03
Request 1	20060920-**04**
Request 2	20060920-05
Sample 4	20060920-**06**
Sample 5	20060920-07

Examining the codes in the table, it is impossible to see which unique codes identify the beginning (i.e. code 20060920-04) and end (i.e. code 20060920-05) of request generation and the resumption of sample generation (i.e. code 20060920-06), as the numbering of the unique codes is sequential.

9.4 Use of Unique Code Masks

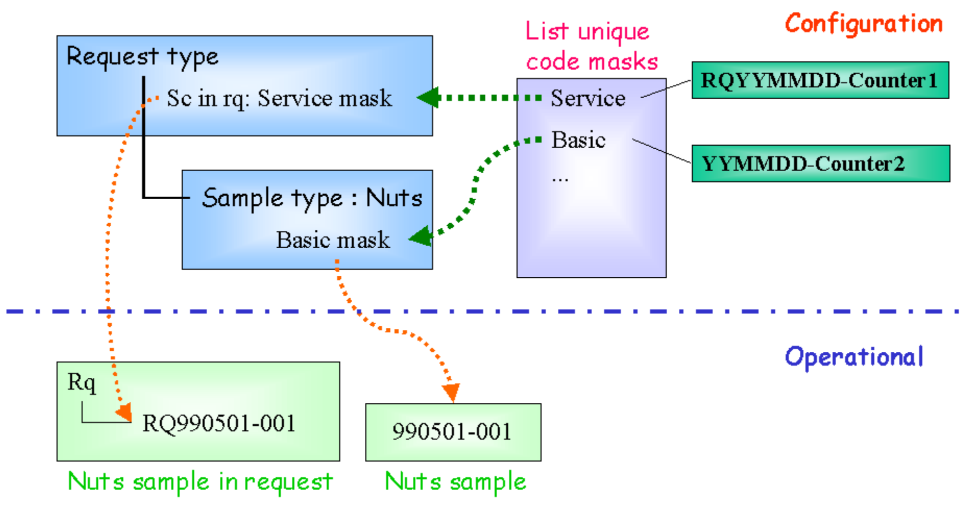
Assigning unique code masks

In the properties of the unique code mask itself, it can be specified for which object type the unique code mask will be used as the default mask (sample code, request code or worksheet). This setting is used system-wide for all sample codes, request codes and worksheet codes.

However, the default mask can be overruled for individual sample types or request types. In the sample type / request type / worksheet type properties, it can be specified which code mask will be used for the samples/ requests of the current sample type / request type / worksheet type. This permits the definition of a different code mask for each sample type / request type / Worksheet type.

For samples belonging to the request, a special code mask can be applied. This mask, set on the request type level, always overrules the code mask set for the sample type itself. Typically it includes a reference to the request.

The picture below shows how the unique code for a sample is generated.



Editing the generated sample / request ID during the creation process is possible. This way, the generated ID can be extended with a number of fields during the sample / request login process. In this case, uniqueness of the ID is checked immediately.

Note that the column in the sample creation / request creation window containing the sample code / request code must be editable!

Gaps in sample codes / request codes

Special attention should be paid to the generation of IDs. Upon creation of a sample (request), its ID is generated. When this ID generation is not followed by a Save action to the database, the ID is lost. The counter is not reset to the previous value. The ID is not attributed to the next sample (request) that is created. This can cause gaps in the sample list / request list. Note that these gaps are traced by the system in the table utucaudittrail.

9.5 Customizing Week-Number Schemes

Although week numbers are often used to uniquely identify samples and/or requests, the definition of the week number itself is subject to interpretation. Consequently, in addition to the ISO or Oracle definition, many laboratory-specific definitions are in use.

Therefore, a special table, **utweeknr**, has been added to the database. In this table, the day numbers and week numbers are maintained for all future dates up to December 31, 2016. Week-number values can be filled out in the table using freehand SQL in accordance with the customer's week definition. Code masks including the week number can then be built based on this table.

10 Formatting and Conversion of Results

10.1 Introduction

Results entered in Unilab undergo a number of conversions.

Entered values are first converted into numeric values. These values are subsequently formatted according to the specified format. The formatted value, a string value, is displayed on the screen.

For some results, an additional conversion is performed. For instance:

- The conversion of method results into parameter results
- The conversion of values that exceed measurement ranges.

10.2 Formatting of Results

For each result in Unilab, the data type and format can be specified in the corresponding object definition.

The data type is only checked for info fields. For other objects, it only serves to indicate/limit the possible formats. The format is used to format the entered value.

Storage of results in Unilab

Most results are stored in 2 different formats in the database: once as a (literal) string and once as a (binary) number (in double precision). This applies to:

- Parameter-group results
- Parameter results
- Method results
- Method-cell values.

Some input values are only stored as string values, e.g. for info-field values. Other variables, such as specifications, are only stored as a numeric value.

The string field of a result holds the formatted value. This is the value displayed in the GUI of an application or within reports. The numeric value is not formatted. This value is used for calculations.

Formatting mechanism

The formatting mechanism depends on the format of the entered value. The entered value can be:

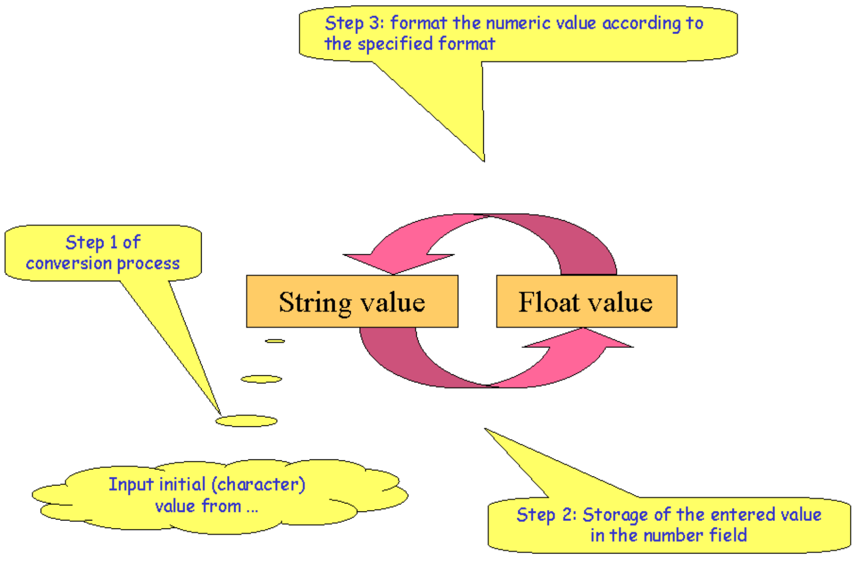
- A numeric value
- A numeric value preceded by a leading sign (<<, <, >, >>, ~)
- A pure string.

The formatting mechanism for each of these values is explained in the following paragraphs.

Formatting the numeric value

The entered value is initially stored in the string field. The value in the string field is converted into a number. This value is stored in the numeric field. The value is then formatted according to the specified format. The string field is updated accordingly.

The picture below shows how to format a numeric value.

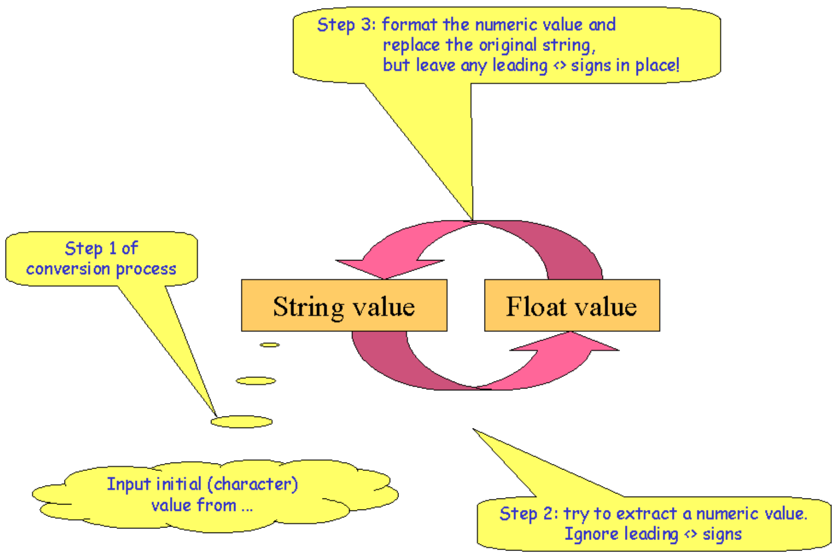

Step 3: format the numeric value according to the specified format
Step 1 of conversion process
String value
Float value
Input initial (character) value from ...
Step 2: Storage of the entered value in the number field

Example: When the user enters **15.542**, the numeric value stored is **15.542**. If the format specifies only a 2-digit (decimal) precision, the string value will become **15.54**.

Formatting strings with leading signs

Some results are numbers preceded by leading signs (e.g. **< 10, ~1000, >>100**). In this case, the system will store the text in the string field and attempt to convert it into a number. This conversion can come about by removing the leading special characters (**<**, **>**, **<<**, **>>**, **~**) and formatting the remaining number as specified in the format.

The picture below shows how to format results with leading signs.


Step 3: format the numeric value and replace the original string, but leave any leading <> signs in place!
Step 1 of conversion process
String value
Float value
Input initial (character) value from ...
Step 2: try to extract a numeric value. Ignore leading <> signs

Example: When the user enters <10, the numeric value stored is 10. If the format specifies only a 2-digit (decimal) precision, the string value will become <10.00.

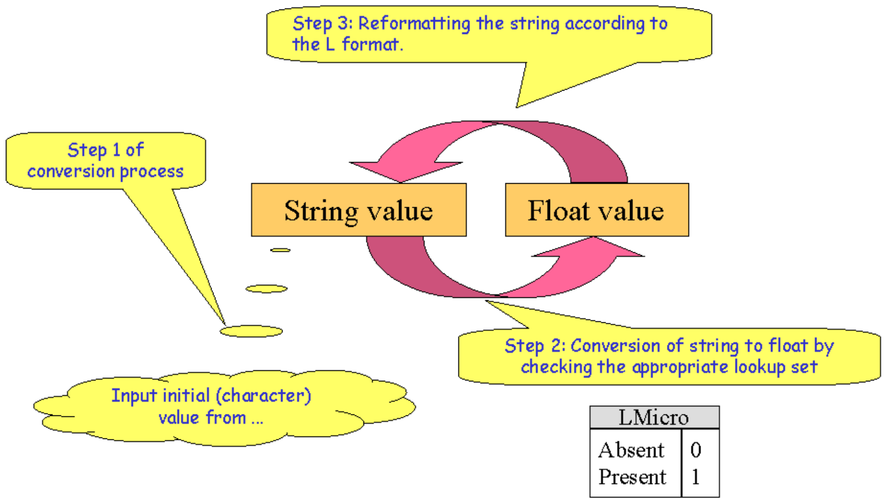
Formatting pure strings

Some entered results are pure strings. For example, this is the case for parameters such as **Salmonella** (possible values **Present** or **Absent**), **Taste** (**Good** or **Bad**) or **Color** (**Blue**, **Green**, **Yellow**).

The concept of lookup values has been introduced to deal with such results. A lookup value is an association of a numeric value with a specific alphanumeric value. When a pure string in entered, the system checks the lookup value to determine the corresponding numeric value. For such results, the format **L** is used (**Lookup** format).

Lookup values can be grouped into lookup sets. This allows associating one numeric value with several alphanumeric values. The appropriate lookup set to be used for a specific parameter is determined by appending the set name to the **L** format identifier.

The picture below shows how to format a pure string.


Step 3: Reformatting the string according to the L format.
Step 1 of conversion process
String value
Float value
Step 2: Conversion of string to float by checking the appropriate lookup set
Input initial (character) value from ...
LMicro
Absent 0
Present 1

Example: Two lookup sets can be specified (one for microbiological analyses and one for sensorial analyses) with the following lookup values:

Micro		**Sensorial**	
Absent	0	Very good	5
Present	1	Good	4
		Neutral	3
		Bad	2
		Very bad	1

The lookup set can be configured for a specific parameter (for instance **Salmonella**) by specifying the format (for instance **Lmicro**).

Supported formats

The supported formats for results (parameter groups, parameters, methods and method cells) and/or for info fields are listed in the table below.

Format Type	**Code**	**Description**	**Results**	**Info Fields**
Float	F	Applies the normal mathematical rounding principles to truncate the number to a given (decimal) precision.	☑	☑
Integer	I	Truncates the number to an integer value (only supported for numbers in the range $-2^{31}+1$, $+2^{31}-1$ (-2147483647, +2147483647))	☑	☑
Round	R	Rounds the number to a given precision	☑	☑
Significance	r	Only shows the given number of significant digits	☑	☑
Exponential	E	Converts to exponential notation	☑	☑
As-is	f	Does not reformat the entered value	☑	
Date	D	The entered value is considered a date	☑	☑
Fixed-length Text	C	Text up to a specified number of characters	☑	☑
Lookup	L	Lookup format	☑	☑
Variable	V	Rounding format that supports user-specifiable	☑	☑

Format Type	Code	Description	Results	Info Fields
		intervals		
Mask	M	Format for Edit masks	☑	☑
Numeric	N	Shows the most accurate numeric value representation corresponding to the original string value that was entered.	☑	☑

For each format, specific details can be given to indicate the number of digits or the precision.

Note

Although the formats **Date** and **Fixed-length text** can be used for methods, they should be used with extreme care since there is NO conversion to a floating-point value for these cases. They are handled as pure text fields, even if they contain data resembling a normal number.

Float format type

For the **Float** format, the following generic structure can be used:

```
F[flags][width][precision]
```

First, a description of each of these details is given. Keep in mind that each component making up the structure is optional.

Flags

The flag directive is always a single character that specifies how the output is justified. It also controls leading signs and blanks.

Flag	Meaning	Default
-	Left justifies the result within the given field width.	Right justifies.
+	Prefixes the output value with a sign (+ or -) if the output value is of a signed type.	Sign appears only for negative-signed values.
0	If the width is prefixed with 0, zeros are added until the minimum width is reached. If both 0 and – appear, the 0 is ignored.	No padding

Width

The width specifies the minimum/maximum number of characters in the result output.

If the number of digits before the decimal sign is less than the specified width, blanks are added to the left, if the '-' flag (for left justification) is present, until the minimum width is reached. There is no padding of blanks at the end of the result.

If the resulting output exceeds the limit imposed by 'width', the resulting output will yield '#' characters. The internal binary number representation is not influenced by this truncation!

If no width is specified, neither minimum nor maximum is applied!

Precision

The precision indicates the number of decimal places or the number of significant digits. The precision is always preceded by a period (.). The value is automatically rounded to the appropriate number of digits. The precision must always be specified using the dot (.), independent of the decimal separator that is in use.

Note that, if a decimal point appears, at least one digit appears before it. Thus, a result always becomes **0.123** and never **.123**.

The default decimal separator is derived from the **Windows Regional Settings** (specified in the Control Panel). Note that all clients must use/apply the same setting (otherwise, a mix of decimal separators will be stored in the database). Of course, the NLS settings in the database must be consistent with the client definition (See Oracle system administrator manuals for more specific details).

The table below gives some examples.

Value	Format	Result
150	F6.2	“150.00”
	F+6.2	“######”
	F10.1	“ 150.0”
	F-10.1	“150.0”
	F4.2	“####”
	F4.0	“ 150”
	F+08.2	“+0150.00”
	F08.2	”00150.00”
	F+8.2	“ +150.00”
0.166	F2.1	##
	F4.2	“0.17”
	F4.0	“ 0”

Integer format type

For the **Integer** format, the following generic structure can be used:

```
I[width][.precision]
```

The width specifies the maximum number of digits in the output.

The precision specifies the minimum number of digits to be printed. If the number of digits in the result is less than *precision*, the output value is padded on the left with zeros. The value is not truncated when the number of digits exceeds precision.

The flag directive is always a single character that specifies how the output is justified. It also controls leading signs and blanks.

Note

This format is only supported for numbers in the range $–2^{31}+1$, $+2^{31}-1$ [-2147483647, +2147483647].

The table below lists some examples.

Value	Format	Result
1300	I6.2	“ 1300”
	I10.2	“ 1300”

	I2	“##”
10.125	I4.2	” 10”
	I4.3	“ 010”
	I-4.2	“10 ”
	I+4.2	“ +10”
10.80	I4.2	“ 10”
0.166	I4.2	“ 00”
	I4.1	“ 0”

Round format type

For the **Round** format, only the basis for the rounding can be specified:

```
R[basis]
```

The number will be rounded to the nearest multiple of basis. This basis can be an integer value or a decimal number.

The table below lists some examples.

Value	Format	Result
1300	R1	“1300”
	R.1	“1300.0”
10.125	R.01	”10.13”
	R.25	“10.25”
0.166	R.2	“0.2”
	R.1	“0.2”
	R.01	“0.17”
0.33	R.2	“0.4”

Significance format type

For the **Precision** format, only the number of significant digits can be specified:

```
r[significance]
```

Only the specified number of significant digits will be retained.

The table below lists some examples.

Value	Format	Result
1234.567	r.2	“1200”
	r.3	“1230”
	r.4	”1235”
	r.5	“1234.6”
0.166	r.2	“0.17”
	r.4	“0.1660”
	r.1	“0.2”
1	r.2	“1.0”
0.32	r.1	“0.3”
	r.2	“0.32”
0	Any precision	“0”

Exponential format type

For the **Exponential** format, the following generic structure can be used:

```
E[flags][width][.precision]
```

For a description of each of these details, refer to the section on Float Data Type. Keep in mind that each component of the structure is optional.

There are always 3 digits in the exponent; the flags are the same as for the float format.

The precision specifies the number of digits after the decimal point. The last printed digit is rounded. The width specification limits the maximum number of digits in the output! The precision controls the size of the output (because the decimal position is automatically adjusted).

The flag directive is always a single character that specifies how the output is justified. It also controls leading signs and blanks.

Important remarks:

- If no precision is given, the width must at least be >= 6 (e.g. 1E+001)
- If a precision is given, the width must at least be >= 8 (e.g. 1.0+001).

The table below lists some examples.

Value	**Format**	**Result**
1300	E9.2	“1.30E+003”
	E10.1	“ 1.3E+003”
	E5.2	“#####”
10.125	E9.2	”1.01E+001”
	E6.0	“1E+001”
0.166	E9.2	“1.66E-001”
	E9.0	“ 2E-001”
	E+7.0	“+2E-001”
	E011.2	“001.66E-001”
	E+011.2	“+01.66E-001”
	E-11.2	“1.66E-001“

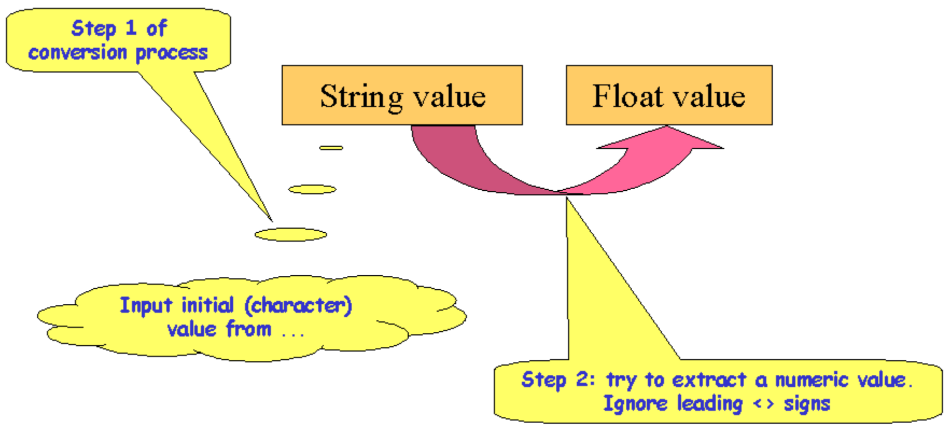
As is format type

For the **As is** format, the following generic structure can be used:

```
f[flags][width][.precision]
```

This is a very special format, because it does not truncate/modify (= format) any input value. However, if this format is used to show a numeric value on the screen (e.g. to show the specs for a parameter), it is considered equal to the F format. Refer to the section on the ‘Float Format’ for more information.

The picture below shows the formatting steps in case of the f format.


Step 1 of conversion process
String value
Float value
Input initial (character) value from ...
Step 2: try to extract a numeric value. Ignore leading <> signs

The figure above depicts the normal formatting steps. The **f** format will execute step 1 and 2 as usual, but will skip step 3 entirely; the original string is not modified in any way! However, if any numerical/float value (like specifications or limits) are retrieved from the DB and have to be displayed, as a string, the float-to-string conversion will be performed as if a **F** format had been specified!

Example: If the parameter **FFA** has format **f2.1** and **7.12** is entered as a result, this value will not be rounded to a precision of one digit (the string result remains "7.12"). But, at the same time, the (numeric) specs will be shown with a precision of one digit.

Date format type

For the **Date** format, the following generic structure can be used:

```
D[date format]
```

Any valid Oracle date format (preceded with a capital 'D') can be specified. The format may include time details.

If a [date format] is given, the date must match the Oracle date mask. The date will be validated by the database. If no date format is specified, the default validation will be used, in which validation is attempted regardless of whether time details are present or not.. Oracle allows almost any special character as a delimiter between the date-parts (e.g. in '01%02%1998', '%' can be almost any special character).

Important

The first 'D' is NOT part of the Oracle date format.

The table below lists some examples.

Input Value	Format	Result
13-01-98	DDD/MM/YYYY	"13/01/0098"
13-01-1998	DMM/DD/YYYY	"01/13/1998"
13/01/98	DDD/MM/YY HH24:MI:SS	Error: time indication in format but no time found in input value
7/4/2001	DDD/MM/YYYY	"07/04/2001"
7/4/2001 09:20:00	DDD/MM/YY HH24:MI:SS	"07/04/01 09:20:00"

7/4/2001	D	“07/04/2001 13:30”

Note

The format **D** reflects the default date format (assume we used DD/MM/YYYY HH24:MI as date and time formats in **Windows settings**).

Important

For some Oracle date formats, there is no appropriate mapping to an equivalent Windows date format – used to display dates in the client applications. The same holds for Windows date formats mapped to Oracle date formats. This phenomenon can create very strange errors.

Therefore, whenever specifying a date format, be sure to cross-check the Windows and Oracle documentation to verify if there exists an equivalent date format in both systems – including an appropriate mapping function.

Character text format type

For the **Fixed Length Text** format, the following generic structure can be used:

```
C[width]
```

The width specifies the maximum number of characters in the result output. If the input string is too short, no padding is done.

If width is not specified, no truncation is performed unless the text exceeds the maximum field length. Info fields have a maximum field length of 1800 characters; other fields are limited to 40 characters.

The table below lists some examples.

Value	Format	Result
1300	C2	“13”
Belgium	C5	“Belgi”
Belgium	C	“Belgium”
Belgium	C-10	“ Belgium”

Lookup format type

The **Lookup** format can be used to force an immediate check in the lookup-table for a corresponding float value. This format has no visible effects, but the numeric value is filled in with the corresponding value from the lookup table.

```
L
```

The **Lookup** format can be used to implement customized pick lists. As the number of entries in a lookup table is (virtually) unlimited, the lookup format supports any number of pick lists identified with a name. This includes the name of the pick list to be used. The current list of lookup values is called **L**.

Lookup list(s) are defined and maintained in the **Configuration** application. Lookup lists can be created or modified by activating the **Lookup** command.

Variable format

The formatting of an entered value often depends on the actual value itself. Some examples are given below:

- In the range 0-10 (**[0,10[**), the format **R.1** must be applied
- In the range 10-100 (**[10,100[**), the format **R1** must be applied
- In the range above 100 (**[100**), the format **R10** must be applied.

This can be handled using the **Variable** format. For the **Variable** format, the following generic structure can be used:

```
V[range name]
```

The range name refers to the name of the range as specified in the **utvformat** table.

Example: For a parameter **pa1**, the format is set to **Vdefault**. In the **utvformat** table, the following ranges are specified for this format:

range _name	seq	range_min _boundary	range _min	range _max	range_max _boundary	format	
default	1	[	0	10	[	R.1	interval [0,10[
default	2	[	10	100	[	R1	interval [10,100[
default	3	[	100			R10	interval >100

The filled-out results will be formatted as displayed in the following table:

Value	Applied Format	Result
1.23	R.1	"1.2"
10	R1	"10"
10.23	R1	"10"
10.99	R1	"11"
11.23	R1	"11"
100	R10	"100"
111.23	R10	"110"

Mask format

The mask format (M) is a special format to be used in edit masks (for instance, info fields and method cells). Edit masks provide restricted data input, as well as formatted data output. It supplies visual clues about the type of data being entered or displayed.

For an input mask, using the M format, each character position in the edit mask maps to either a placeholder of a specified type or a literal character. Literal characters, or literals, can give visual clues about the type of data being used. For example, the parentheses surrounding the area code of a telephone number are literals: (206). The input mask prevents the user from entering invalid characters into the control.

The table below lists the mask characters for M format.

Mask character	Description
#	Digit placeholder. (0-9)
.	Decimal placeholder. The actual character used is the one specified as the decimal placeholder in your international settings. This character is treated as a literal for masking purposes.
,	Thousands separator. The actual character used is the one specified

	as the thousands separator in your international settings. This character is treated as a literal for masking purposes.
:	Time separator. The actual character used is the one specified as the time separator in your international settings. This character is treated as a literal for masking purposes.
/	Date separator. The actual character used is the one specified as the date separator in your international settings. This character is treated as a literal for masking purposes.
A	Alphanumeric character placeholder (0-9 and a-Z).
&	Character placeholder. Valid values for this placeholder are ANSI characters in the following ranges: 32-126 and 128-255.
?	Alphabetic placeholder (a-Z).
>	Alphabetic placeholder, but forces them to Uppercase chars (A-Z)
<	Alphabetic placeholder, but forces them to lowercase (a-z)
\	Treats the next character in the mask string as a literal. This allows you to include the '#', '&', 'A', '<', '>',and '?' characters in the mask. This character is treated as a literal for masking purposes.
Literal	All other symbols are displayed as literals; that is, as themselves.

The table below lists some examples.

Value	**Format**	**Result**
9402	MB-####	"B-9402"
belgium	M>AAAAAAAAAA	"Belgium___"
123456789	M###.###.###	"123.456.789"
3254324748	Mtel. +## ## ### ###	"tel +32 54 324 748"
	M+## ## ### ###	"+32 54 324 748"

10.3 Conversion of Input Strings to Numeric Values

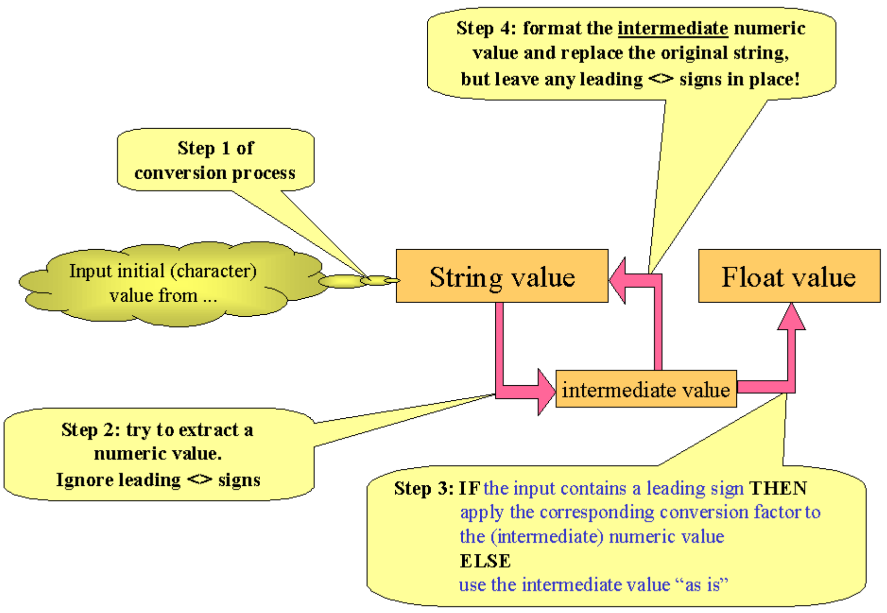
Specific conversion rules are often applied when leading signs are used. These rules define how the results must be converted for use in calculations, statistical analysis or charts.

10.3.1 Conversion of Strings with Leading Signs

Often, results are entered starting with a leading sign. The aforementioned formatting rules apply. However, using the numerical part of the entered result in the float value might cause misleading results should further calculations or statistical analysis be performed.

For example, consider an environmental laboratory. Upon analyzing a waste-water sample, the content of a certain compound appears to be below the determination limit of the instrument. The returned result is, for instance, **<10**. Reporting a numeric value of **10** is not correct (it should be less). Neither should the value be set to 0, because analysts cannot state with absolute certainty that the compound is absent from the sample, but only that, at the current content, its presence cannot be determined with the instrument at hand. In such cases, laboratories apply a conversion to the obtained result. For instance, the result is multiplied by **0.5**.

The following figure illustrates the conversion process to obtain a formatted value from an initial input string.


Step 4: format the intermediate numeric value and replace the original string, but leave any leading <> signs in place!
Step 1 of conversion process
Input initial (character) value from ...
String value
Float value
intermediate value
Step 2: try to extract a numeric value. Ignore leading <> signs
Step 3: IF the input contains a leading sign THEN apply the corresponding conversion factor to the (intermediate) numeric value ELSE use the intermediate value "as is"

On input of a string value, Unilab first attempts to extract the numeric value by ignoring the leading signs. If leading signs were present, a conversion factor is applied to the intermediate numeric value. These conversion factors are defined system-wide for each of the leading signs individually. If no leading signs were present, the numeric value is stored as such. The intermediate numeric value (not the converted numeric value) is then formatted according to the specified format. The original string is then replaced with the formatted string. The leading signs are kept in place.

Note

This string-to-numeric conversion applies to any input (including, for example, method cells and info fields)!

Example for the conversion of strings with leading signs

- Conversion factor for < sign: **Conv_factor_< = *0.5**
- Format: **F4.1**

The value **< 10** is entered. The following steps occur:

Step	String	Float	Comments
Step 1	<10		The value is stored in the String value column
Step 2	<10	(10)	The numeric value is extracted and stored in float column (intermediate value)
Step 3	<10	5	The conversion is applied to the float value
Step 4	<10.0	5	The intermediate numeric value is formatted to obtain the final string value.

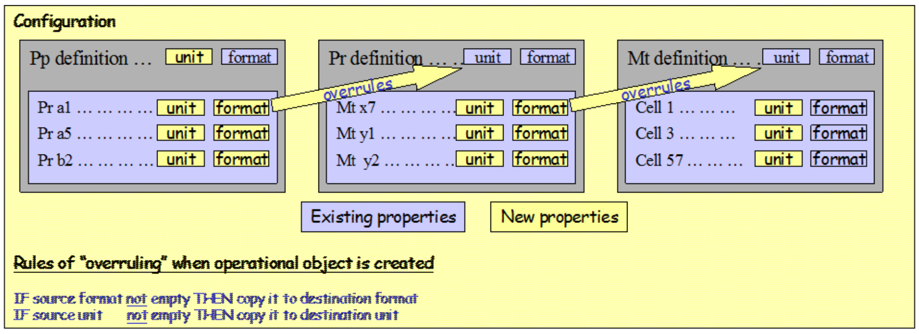
10.3.2 Conversion of Units and Results

Because several method cells, methods and parameters can have their own accuracies and units, rules of conversion between the various units must be defined. These conversion rules are important, for example, to measure a parameter using several methods, each with its own accuracy, format and unit.

Unit and format as standard attributes for used parameters and methods

Unit and formats can be defined in the standard attributes of the used objects (used parameters and used methods). This means that it is possible to overrule the parameter unit and format on the used parameter level according to the normal overruling mechanism. Likewise, it is possible to overrule the method unit and format on the used method level.

The picture below shows the overruling scheme for units and formats.


Configuration
Pp definition … unit format
Pr a1 … … … … unit format
Pr a5 … … … … unit format
Pr b2 … … … … unit format
overrules
Pr definition … … unit format
Mt x7 … … … … unit format
Mt y1 … … … … unit format
Mt y2 … … … … unit format
overrules
Mt definition … … unit format
Cell 1 … … … unit format
Cell 3 … … … unit format
Cell 57 … … … unit format
Existing properties
New properties
Rules of "overruling" when operational object is created
IF source format not empty THEN copy it to destination format
IF source unit not empty THEN copy it to destination unit

Unit as method-cell property

The unit is specified in the method-cell properties dialog for the following types of method cells: calculated field, combo box, drop-down list, edit field, edit mask, list box, spin box, table. Although the unit, format and result of a table method cell can be specified in configuration, these properties cannot be seen in the operational application.

The unit of a cell cannot be modified in the operational applications.

The unit, if specified, is placed between brackets after the method-cell description in the method sheet, except for table method cells.

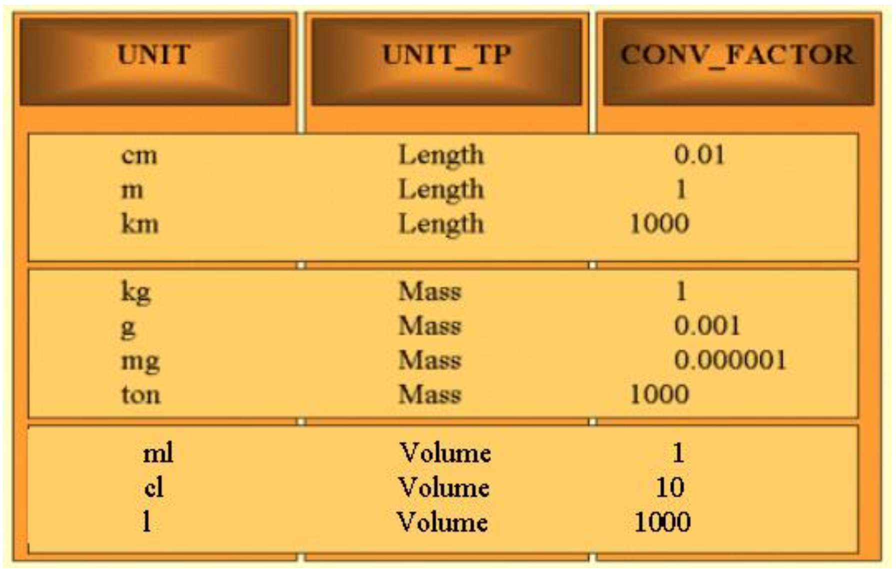
Define list of possible units and conversion factors

Unit and conversion definitions must be set in the Configuration application with the Edit Table functionality.

Each unit belongs to a specific unit type and uses a specific conversion factor. The conversion factor is a multiplication factor that can be either positive or negative, but never zero. The conversion factor is necessary to permit conversion between the various units (e.g. Conversion factor between meters and kilometers is 1000).

Conversion is feasible only between the units that belong to the same unit type. Units with different conversion factors cannot be converted (e.g. unit conversion is possible between units **cm**, **m**, **km**, because all these units belong to unit type **length**.)

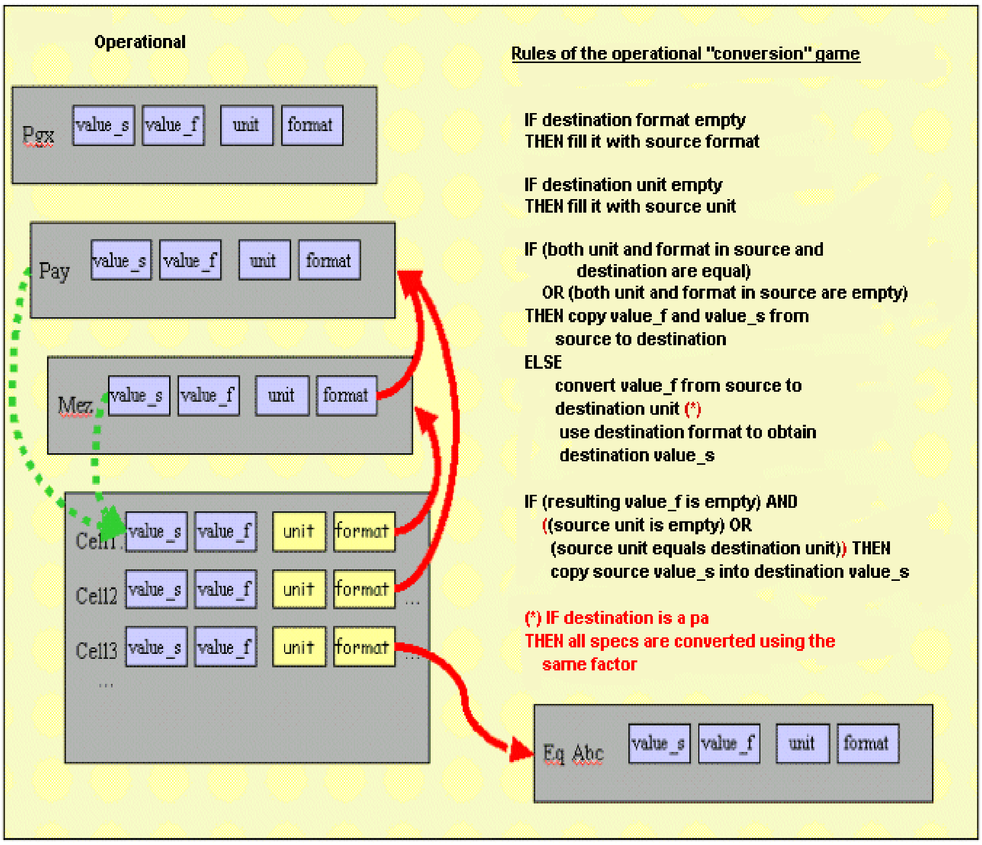
The picture below shows the Unit table UTUNIT.



UNIT	UNIT_TP	CONV_FACTOR
cm	Length	0.01
m	Length	1
km	Length	1000
kg	Mass	1
g	Mass	0.001
mg	Mass	0.000001
ton	Mass	1000
ml	Volume	1
cl	Volume	10
l	Volume	1000

Unit conversions

The picture below shows all unit conversions feasible in Unilab and the conversion rules that apply.



Rules of the operational "conversion" game

```
IF destination format empty
THEN fill it with source format

IF destination unit empty
THEN fill it with source unit

IF (both unit and format in source and
        destination are equal)
   OR (both unit and format in source are empty)
THEN copy value_f and value_s from
      source to destination
ELSE
      convert value_f from source to
      destination unit (*)
      use destination format to obtain
      destination value_s

IF (resulting value_f is empty) AND
   ((source unit is empty) OR
    (source unit equals destination unit)) THEN
    copy source value_s into destination value_s

(*) IF destination is a pa
THEN all specs are converted using the
   same factor
```

The figure also contemplates a conversion from parameter/method result to a method cell. This is possible because method cells can have their input from a parameter or a method result.

If a parameter result is the average of several methods, then the result of these methods is first converted to the unit of the first method before the average is calculated.

Important

Custom functions for parameter-result calculation must be written with unit conversion provided as a “built-in” feature.

Units cannot be converted into units of another unit_type (e.g. km cannot be converted to kg). If this is attempted on the operational level, then the string value (value_s) of the result is set to ‘Conversion Error’ and the float value (value_f) to NULL. This makes it clear to the user that there is a configuration error. This also applies when a parameter result is the average of methods with incompatible units (e.g. average(1km, 1kg) = ‘Conversion Error’).

Example:

A method with value 123.4 and unit ‘cm’ is saved as result of a parameter with unit ‘m’. The float value (value_f) of this parameter is 1 * (conversion_factor ‘m’ / conversion_factor ‘cm’) = 123.4 * 1/100 = 1.234.

An example of unit conversion is provided below:

Source: **Method: unit: cm ; Format F6.2 ; value_f: 123.4 ; value_s: “123.40”**					
Destination: Parameter					
unit (before)	**format (before)**	**value_f**	**value_s**	**unit (after)**	**format (after)**
m	F8.4	1.234	“1.2340”	m	F8.4
g	F8.4	NULL	“Conversion Error”	g	F8.4
NULL	NULL	1.234	“123.40”	cm	F6.2
NULL	F7.3	1.234	“123.400”	cm	F7.3
m	NULL	1.234	“1.23”	m	F6.2

No conversion is performed if the ‘from’ and ‘to’ objects have the same unit and format.

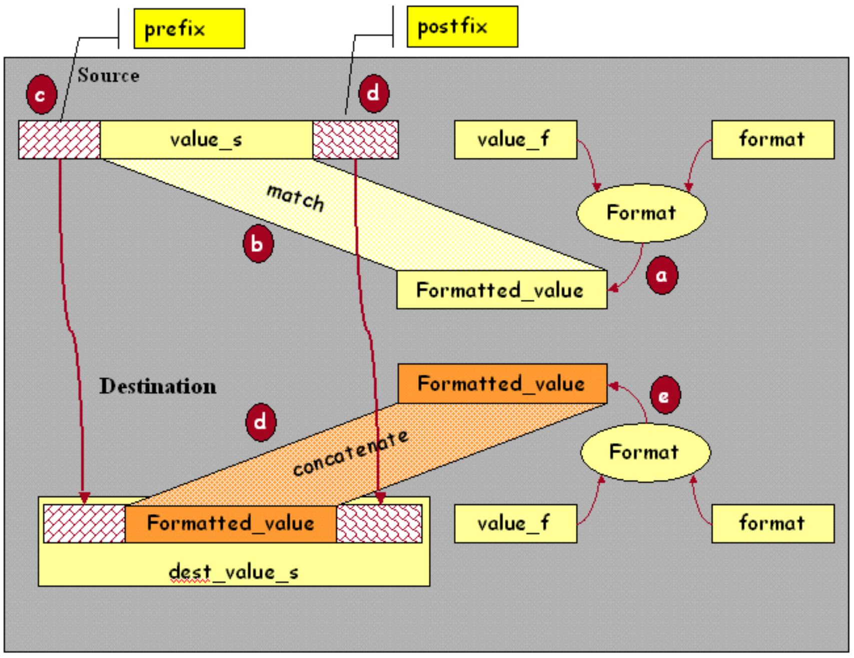
If the unit of an object B is not specified and the value of an object A is saved as the result of object B, then the value and unit are copied to object B. If the format of object B is not specified, then the format of object A is also copied.

Example:

The method result has a float format and cm as its unit. This method result is saved as the parameter result. No format and unit have been specified for this parameter result (format=NULL, unit=NULL). Because the format and the unit of the parameter result are not indicated, the float format and unit (i.e. cm) of the method result are also copied.

The string value (value_s) of object B is defined by its format and its float value (value_f). If the original string value (value_s) (object A) has a prefix (e.g <) or a postfix, the following schema is used.

The picture below shows the conversion of values with prefix or postfix.

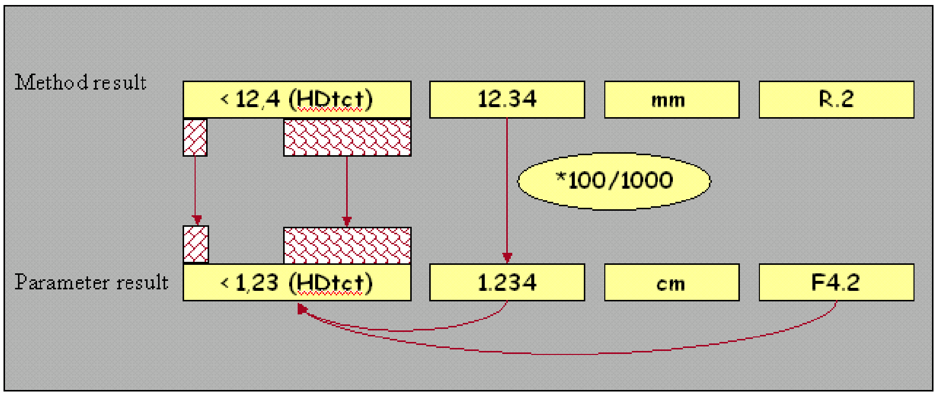


b

Basically, the schema above indicates that the prefix and postfix around the formatted value in the value_s are fetched and copied around the formatted value on the next level.

If the formatted value cannot be found in the value_s on the primary level (e.g. value_s = 'too small'), then the value_s on the next level is the formatted result based on the value_f and the format (the information is the value_s on the primary level is lost in this case).

The picture below shows an example of unit and conversion.



Unit conversion from method to parameter is applied after the conversion rules (e.g results with leading sign <, >) or measurement-range conversion on the method result.

Example:

A method with value_s (string value) <123.40(LDT) and unit ‘cm’ is saved as result of a parameter with unit ‘m’. The prefix (<) and postfix (LDT) around the formatted method result value in the value_s are fetched and copied around the formatted parameter result value.

The table below provides an example of unit conversions with measurement ranges.

Source: **Method: unit: cm ; Format F6.2 ; value_f: 123.4 ; value_s: “< 123.40 (LDT)”**					
Destination: Parameter					
unit (before)	**format (before)**	**value_f**	**value_s**	**unit (after)**	**format (after)**
m	F8.4	1.234	“< 1.2340 (LDT)”	m	F8.4
g	F8.4	NULL	“Conversion Error”	g	F8.4
NULL	NULL	1.234	“< 123.40 (LDT)”	cm	F6.2
NULL	F7.3	1.234	“< 123.400 (LDT)”	cm	F7.3
m	NULL	1.234	“< 1.23 (LDT)”	m	F6.2

note that there can be no match between value_s and formatted value_f. This can happen when the system setting **CONV_FACTOR_<** is applied or when some custom code is used to fill in the value_s. If the formatted value cannot be found in the value_s on the primary level (e.g. value_s = ‘too small’), then Unilab will not make any further attempts.

Please refer to “Special Cases” below for details on how to handle such cases.

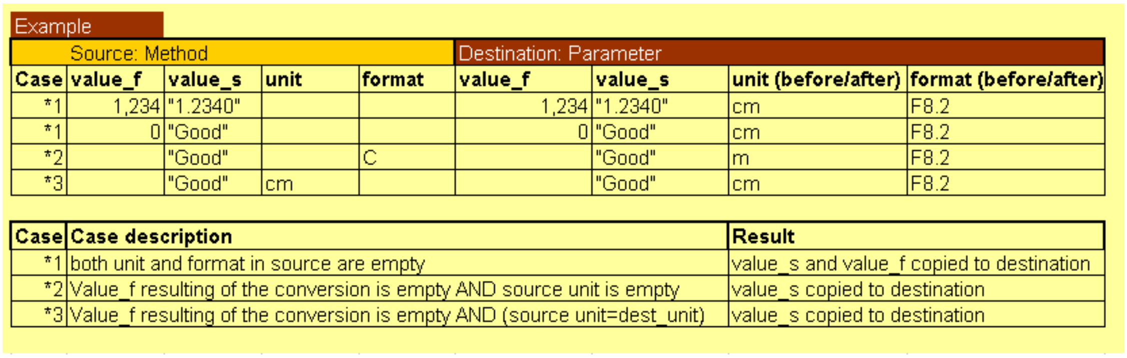
Conversions between units are performed only with multiplication factors. Conversion functions cannot be specified. E.g.: it is not possible to convert degrees Celsius into Fahrenheit.

No unit conversion is applied for the calculation of calculated method cells when ‘simple Calculation’ or ‘Visual Basic script’ is chosen. This means that the calculation formally must be modified if the unit of a method cell is changed. Keep in mind that the unit of method cells cannot be modified in the operational applications. Custom functions can take account of the units of cells. The normal conversion rules apply between the calculated cell and the method value, if the calculated cell is set as the method result. Therefore, there is no problem if the unit of the method value is overruled on the parameter-method level. Evidently, the unit of the calculated field must be set if one wants the result to be converted into the unit of the method.

The conversion factor is also applied to equipment constants for cell input and output. The Unit of an equipment constant can be specified in Define Equipment. The unit-conversion rules between equipment constants and method cells are the same as for other objects.

Special Cases

In some very specific cases, any conversion and formatting can be bypassed/avoided by using ad-hoc configurations illustrated in the matrix below:



Example

Source: Method					Destination: Parameter			
Case	**value_f**	**value_s**	**unit**	**format**	**value_f**	**value_s**	**unit (before/after)**	**format (before/after)**
*1	1,234	"1.2340"			1,234	"1.2340"	cm	F8.2
*1	0	"Good"			0	"Good"	cm	F8.2
*2		"Good"		C		"Good"	m	F8.2
*3		"Good"	cm			"Good"	cm	F8.2

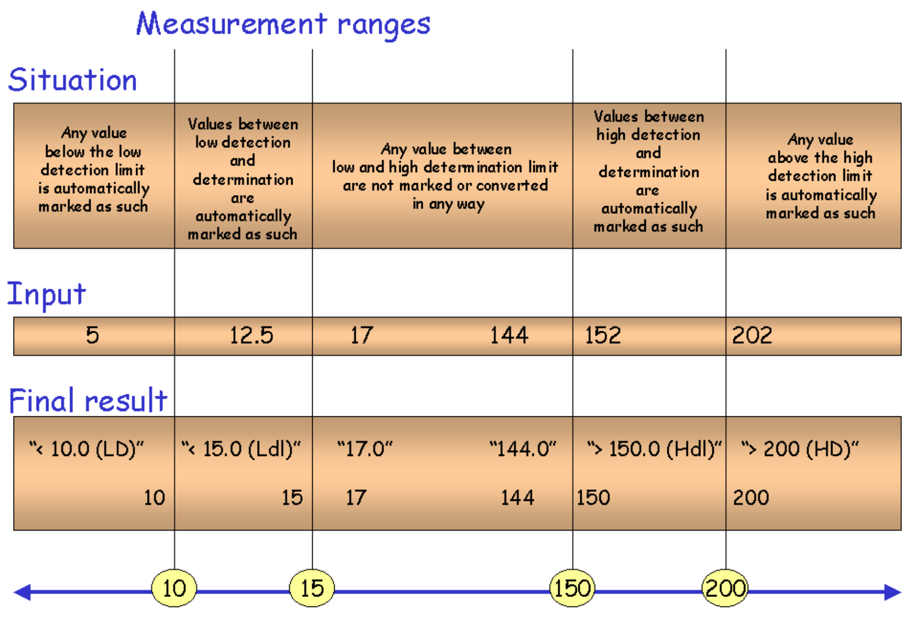
Case	**Case description**	**Result**
*1	both unit and format in source are empty	value_s and value_f copied to destination
*2	Value_f resulting of the conversion is empty AND source unit is empty	value_s copied to destination
*3	Value_f resulting of the conversion is empty AND (source unit=dest_unit)	value_s copied to destination

Note that the key elements to avoid the formatting and conversion are the empty elements in the matrix (empty value_f, empty unit or empty format in the source object).

10.4 Measurement Ranges

For method cells, the client application always applies the normal conversion and formatting rules first. However, when the **Component** field of the method cell is not empty, a number of additional formatting rules will also be applied.

The value entered in the method cell will be checked against the measurement ranges specified for that specific component. The entered value will be formatted correspondingly. The picture below shows the formatting rules in the case of measurement ranges.


Measurement ranges
Situation
Any value below the low detection limit is automatically marked as such
Values between low detection and determination are automatically marked as such
Any value between low and high determination limit are not marked or converted in any way
Values between high detection and determination are automatically marked as such
Any value above the high detection limit is automatically marked as such
Input
5
12.5
17
144
152
202
Final result
"< 10.0 (LD)"
10
"< 15.0 (Ldl)"
15
"17.0"
17
"144.0"
144
"> 150.0 (Hdl)"
150
"> 200 (HD)"
200
10
15
150
200

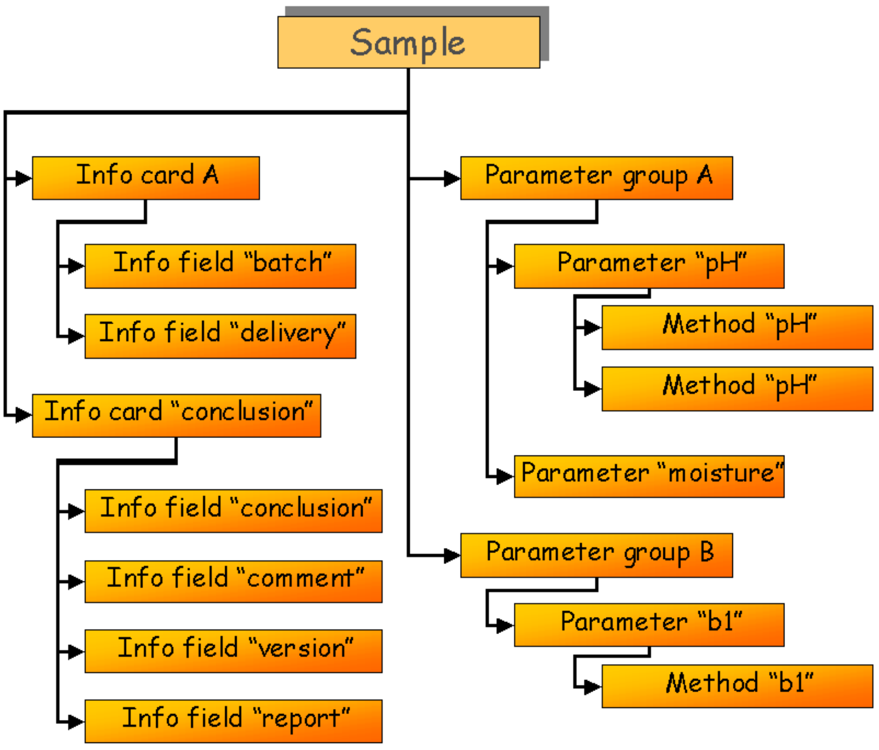
Please note that the measurement ranges are checked only after the normal formatting and conversion rules have been applied. This means that the normal formatting and conversion rules are no longer applied on the result of the check performed on the measurement ranges.

For example, consider an instrument that returns value **12**. With the measurement ranges for that specific component as specified in the picture, the check on measurement ranges returns a string value of **< 15.0 (LDL)** and a float value of **15**. If the normal conversion rules would be applied on the float value, with, for instance, a conversion factor ***0.5**, the final float result would be **7.5** (= **15*0.5**). This result is smaller then the low detection limit (**10**). An additional conversion would ultimately lead to a string value **<10.0(LD)**. To avoid such inconsistencies, no additional formatting and conversion is applied once the measurement ranges have been checked.

11 Frequency Filtering

When a sample is created, parameter groups and info cards are assigned to the sample. Basically, such an assignment corresponds to the association of one object (for instance, a sample) with one or more objects at a lower hierarchical level (for instance, parameter groups).

The picture below shows the tree structure of a sample.



Including the assignment-rule logic into the LIMS system makes it much easier and simpler for an analyst to login a sample (because the current test plan will be generated automatically).

The same rules apply for each hierarchical level of the data model.

11.1 Basic Assignment Frequencies

Parameter groups are assigned to samples on the basis of the assignment frequencies on the configuration level. Basically, there are several types of assignment frequencies: **Always**, **Sample count** based, **Time based**, **Never** or **Custom function** based. The skip flag can be set to invert a set assignment frequency.

Always

This is the default assignment rule, which implies that the object is always scheduled. When, for instance, the assignment frequency of a method to a parameter is set to **Always**, the method will always be assigned automatically whenever the parameter results must be determined.

Sample count based

The scheduling of the object is based on the sample count. For instance, a parameter **pH** can be scheduled once every five samples.

The “sample-count-based” assignment frequency also offers the possibility to make the assignment frequency of an object dependent on certain info-field values.

Example: A specific analysis is performed only once for each supplier delivery. The assignment frequency of the corresponding parameter is made dependent on the value of the info field **Supplier**: each time the value in the info field **Supplier** is modified, which corresponds to a delivery by another supplier, the parameter is scheduled.

Time based

The object is scheduled at regular time intervals. Note that 7 days is different from 1 week and that 24 hours is not the same as 1 day (even though, in both cases, it is the same interval). An object can be also scheduled on a fixed day of each month (or even a fixed day of a specific month of each year).

Never

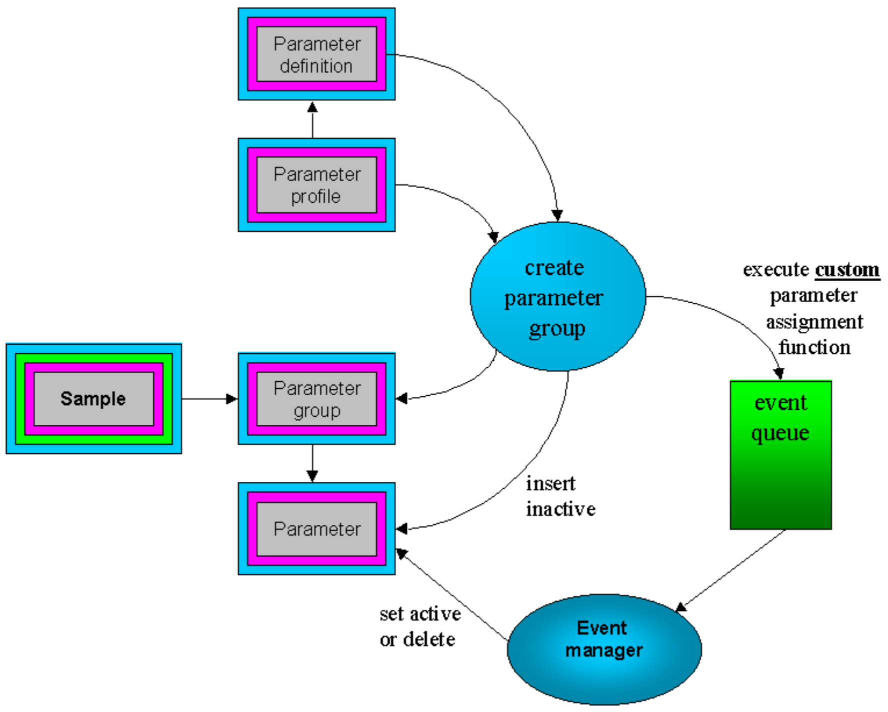
The object is never assigned automatically. However, it can be assigned manually. On the operational level, the user will be able to select the object from the assignment list. When the assignment frequency of the parameter **pH** in the parameter profile **Chemical** is set to **Never**, then this parameter will never be assigned automatically upon sample creation. However, the analyst will be able to select the parameter **pH** from a parameter assignment list on the operational level for manual assignment.

Custom function

If none of the other assignment frequencies is suitable, the application manager can write their own PL/SQL function to define the assignment frequency.

The way custom assignment frequencies are handled by the Event Manager is displayed in the picture below. All used objects on a lower hierarchical level are first assigned as inactive. During the execution of the custom frequency function, the assignment of the objects is evaluated. The objects are either deleted or activated.

The picture below shows how to create a parameter group.



Skip flag

The **Skip** flag can be used to invert a set assignment rule. When an object is assigned with frequency **Skip once per 5 samples,** this implies that the object is assigned to four out of five samples. Note that **skip Always** is the same as **Never**.

11.2 Manual Assignment

The automatic assignment of parameter groups to samples is very convenient in an environment that applies fixed test plans. Note that these test plans are not fixed in the sense that they are always statically the same, but in the sense that the rules are predefined.

Regardless of how elaborate the test plans may be, occasionally some parameter groups will have to be assigned manually. Manual assignment of analyses is even more the rule than the exception in service laboratories, due to the fact that these laboratories work on a request basis. In this case, the requested analyses are assigned “ad hoc” upon sample login.

Unilab provides extensive features to cope with the “ad hoc” assignment of parameter groups (or other objects).

Selection list for manual assignment

The selection list an analyst gets on the operational level for assigning objects is obtained from the configuration level. When, for instance, a parameter group is manually assigned to a sample, by default, the Unilab system will present the restricted list of all the parameter groups related to the parameter-profile list of the corresponding sample type. This list includes those parameter profiles for which the assignment frequency is set to **Never**.

However, in some cases, using a sample-type-based assignment list turns out to be too restrictive. This drawback can be eliminated at the configuration level simply by setting the **Assign any parameter profile** standard property flag for the given sample type. In this case, the complete list of all configured parameter profiles is shown on the manual parameter-profile assignment.

Interactive search functionality

Some flexibility is provided on an operational level by means of the interactive search functionality on the list of configured parameter profiles. This makes it possible for the analyst to restrict the selection list, using search clues including starting characters, parameter-profile attributes, as well as sample-type group-key values. The list can even be restricted to those parameter groups that already have been assigned to the current sample.

Assignment of the objects on a lower hierarchical level

By default, the manual assignment of a parameter group to a sample corresponds to the creation of the parameter group. This means that all objects on a lower hierarchical level (parameters and methods) are assigned with the specified assignment frequencies. Unilab offers nevertheless the possibility to confirm the assignment of the objects on the lower hierarchical level. In this manner, the set assignment frequencies can be overruled.

In the case of method assignment, an extra feature is built into the assignment logic. This extra feature entails that, in many cases, the information on how to proceed to obtain test information (= which test method to use) is linked not directly to the parameter definition itself, but rather to the parameter profile, one level up in the hierarchy.

Water quality control illustrates the need for this information at the parameter-profile level. Some of the parameters measured in the context of waste-water quality are identical to those used for the assessment of drinking-water quality. Nevertheless, the ranges of the measured concentrations, as well as the specs, can differ significantly. They can differ so much that completely different measurement methods need to be used. In fact, these differences are expressed by using different parameter profiles: one for waste water, another one for drinking water. It is possible to associate to a parameter in a parameter profile not only a frequency and specs, but also a specific test method (including the number of times the method needs to be executed).

Assign full test plan

The **Assign full test plan** option permits assigning a specific test plan to a set of selected samples. The test plan is the list of parameter profiles, parameters and methods configured for the corresponding sample type.

This function permits the assignment of all parameter groups / parameters / methods that were not assigned automatically after evaluating the assignment frequencies. It actually allows overruling the configured assignment frequencies.

As for the objects that are assigned, two options are available: insert or create.

Upon creation of a parameter group, the assignment frequencies are applied to the parameter and method level. On insert, only the parameter group is assigned. None of the parameters or methods are assigned automatically.

11.3 Sample Type Based Assignment Frequencies

For parameters of methods, it is possible that the assignment frequency has to be monitored at the sample type level. Consider the following example: the chloride content is determined in water that is coming into the factory and waste water leaving the factory. The analysis has to be performed once every two days on each of the samples. The sample of incoming water arrives in the morning; the waste-water sample arrives in the afternoon. The following test plan is desired for the chloride determination:

	Monday	**Tuesday**	**Wednesday**	**Thursday**
Incoming water	Yes	No	Yes	No
Waste water	Yes	No	Yes	No

When the assignment frequency of the parameter **Chloride** is not sample type dependent, the following test plan will be obtained:

	Monday	**Tuesday**	**Wednesday**	**Thursday**
Incoming water	Yes	No	Yes	No
Waste water	No	No	No	No

The chloride determination is never performed on the waste water samples. The assignment frequency should be made sample type dependent.

12 21 CFR Part 11 Mode

12.1 Introduction

The US FDA rule on electronic records/electronic signatures, 21 CFR Part 11, states that changing electronic records must not obscure previous data. Version control and audit trail have been put forward as the solutions to deal with this gap. In Unilab, with an Advanced Server license (as of V6.1 SP1; in older versions, this was an additional 21 CFR part 11 license key), version control is present on the main configuration objects.

Changes to operational objects are managed in another way. An operational object is created by using the information relative to a specific version of the (corresponding) configuration object. Only individual properties of operational objects can be modified. These changes are traced in the audit trail. Hence, the concept of version control is not implemented for Unilab operational objects.

Important

Version management is only available on systems with the **SIMATIC IT Unilab Advanced Server** license.

12.2 Structural Overview

When working with different versions of a configuration object, certain issues must be taken into account:

- The concept of **allow modify** and **active** flags is valid. In some specific situations, the use of the **active** flag changes slightly. This is explained in more detail later on in this document.
- The version identification of an object can never be changed.

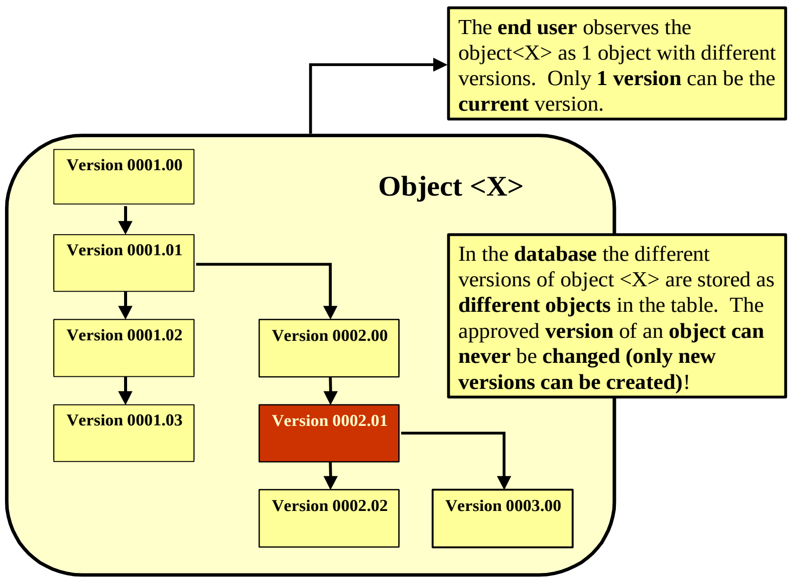
On the configuration level, the primary key of the objects includes the version id of the object.

On the operational level, the version id of the corresponding configuration object type is maintained (as a link) for information/tracing purposes.

Functional access is set so that a version of an object cannot be deleted.

- This is automatically implemented during DB installation on systems with a SIMATIC IT Unilab Advanced Server license.
- In case of archiving, an object version is removed from the original DB after it has been copied to the archive DB or file.

The picture below shows the structural overview of version control.


The **end user** observes the object<X> as 1 object with different versions. Only **1 version** can be the **current** version.
Object <X>
Version 0001.00
Version 0001.01
Version 0001.02
Version 0001.03
Version 0002.00
Version 0002.01
Version 0002.02
Version 0003.00
In the **database** the different versions of object <X> are stored as **different objects** in the table. The approved **version** of an **object can never** be **changed (only new versions can be created)**!

A version number is actually a single string in the database. This permits considerable flexibility in the construction of the version numbering system. The format of a version number can be specified by customizing the package **UNVERSION**.

Tip

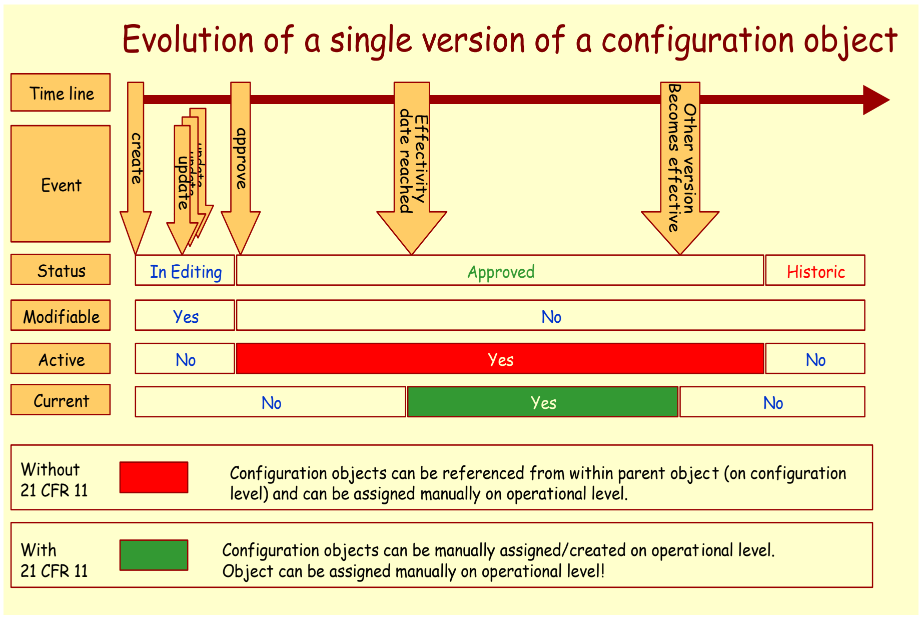
This chapter does not describe how to specify the format of the version number. Therefore, please refer to **Customizing the system** customization guide in the Unilab Product Library.

12.3 The Active and Current Flags

The process for creating new configuration objects does not change substantially when working with version management. However, there are a few minor differences, due to the fact that objects of different types are not always processed in a strictly sequential order. Therefore, it must be possible, for instance, to select or use a reference to any version of an object before it becomes current or even before it is approved.

These differences in the creation process involve the **Active** flag and the **Current** flag.

The picture below shows the evolution of a single version of a configuration object.



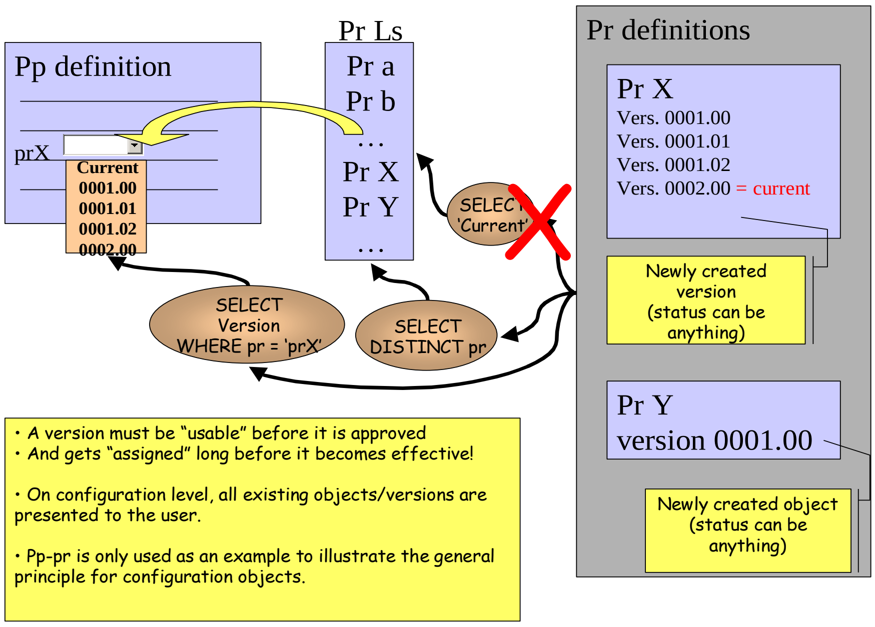
Configuration objects can be referenced from within parent object (on configuration level) and can be assigned manually on operational level.

Configuration objects can be manually assigned/created on operational level. Object can be assigned manually on operational level!

Active flag

The configuration applications for defining an object ignore the **Active** flag of that object. This means that it is possible to create new versions of configuration objects based on active and non-active objects. Also, when using an object definition (as part of another object – for instance, parameter profiles assigned to a sample type), both active and non-active objects can be used for assignment.

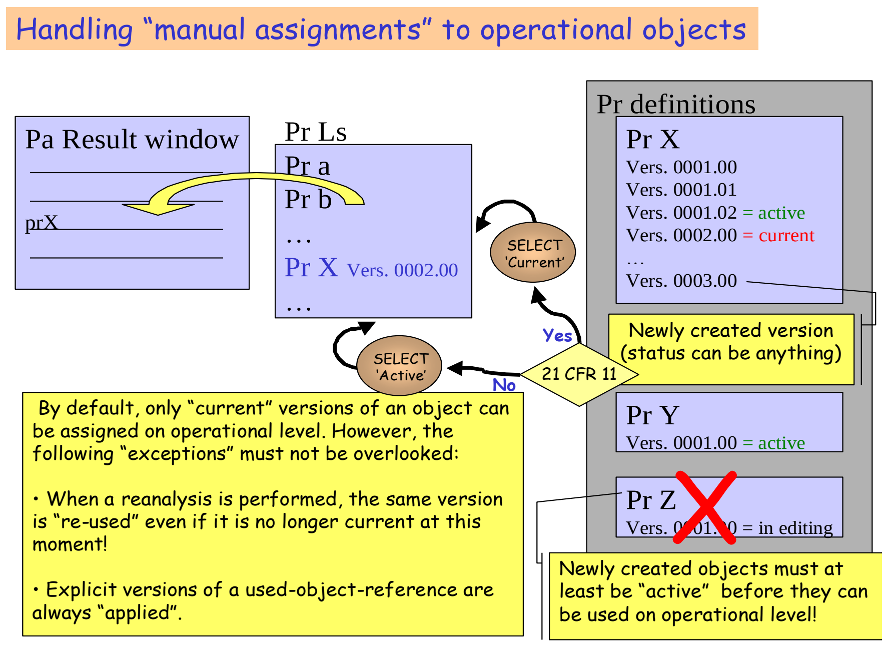
The picture below shows the assignment of objects on the configuration level.



- A version must be "usable" before it is approved
- And gets "assigned" long before it becomes effective!
- On configuration level, all existing objects/versions are presented to the user.
- Pp-pr is only used as an example to illustrate the general principle for configuration objects.

In the operational applications without the 21 CFR Part 11 mode, only configuration objects that are active can be used. Samples (or requests) of sample types (or request types) that are not active cannot be logged on. For example, parameter groups cannot be assigned (to samples) when the parameter-profile definition is not active.

The picture below shows the assignment of objects on the operational level.


Handling "manual assignments" to operational objects
Pa Result window
prX
Pr Ls
Pr a
Pr b
…
Pr X Vers. 0002.00
…
Pr definitions
Pr X
Vers. 0001.00
Vers. 0001.01
Vers. 0001.02 = active
Vers. 0002.00 = current
…
Vers. 0003.00
Newly created version (status can be anything)
SELECT 'Current'
SELECT 'Active'
Yes
No
21 CFR 11
Pr Y
Vers. 0001.00 = active
Pr Z
Vers. 0001.00 = in editing
By default, only "current" versions of an object can be assigned on operational level. However, the following "exceptions" must not be overlooked:
• When a reanalysis is performed, the same version is "re-used" even if it is no longer current at this moment!
• Explicit versions of a used-object-reference are always "applied".
Newly created objects must at least be "active" before they can be used on operational level!

In the operational applications having the 21 CFR Part 11 mode, these rules are more stringent. For example, when a new parameter or method is assigned to an existing sample, the user gets, as before, a list of objects from which a selection must be made. The difference is that, instead of listing the active objects, the system lists all the objects for which there is a current version.

Current flag

In addition to an **Allow_modify** flag and an **Active** flag, a version has a **Current** flag. Only one version of an object can be current.

A version that is not current cannot be used in operational applications, except when it is linked to a current master object. This means that a configuration object with a list of used objects can have a current flag, regardless of the status of its used objects. This implies that, upon creation of the corresponding operational object, the list of used objects will be created as well (according to the set assignment frequencies), even if some of these objects are not active.

The concept of a current version is only applicable to the configuration objects. The operational objects are always created from a specific version of a configuration object. The (configuration) version number of an operational object is part of its standard properties and can never be changed!

Important

The **Current** flag is not controlled by the object status.

12.4 Creation of New Versions

Creation of a new configuration object

By default, when a new configuration object (Example: new sample type) is created, it is always assigned version number 0001.00, assuming 9999.99 is the version format.

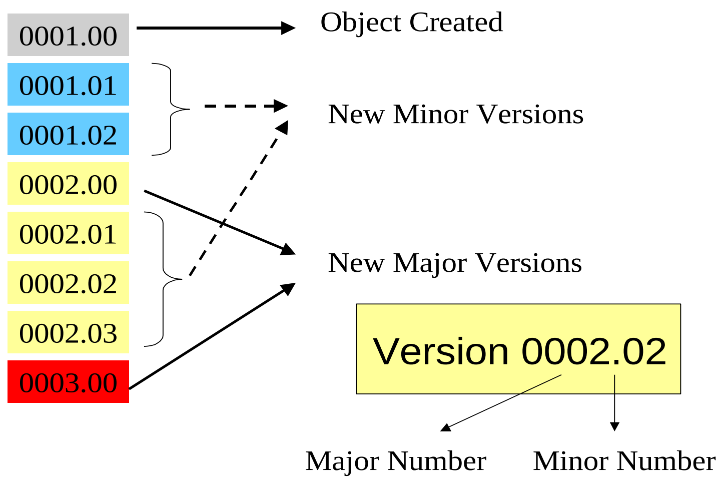
Minor and major version

Configuration objects are always assigned Version 0001.00 upon creation.

When a new version is created for an existing configuration object, it is possible to specify whether it should be a new major or a new minor version of the object. The system will automatically take the next highest major or minor number.

Remember that the possibility of creating a new major version is permitted only to those users who possess the appropriate functional rights to do so.

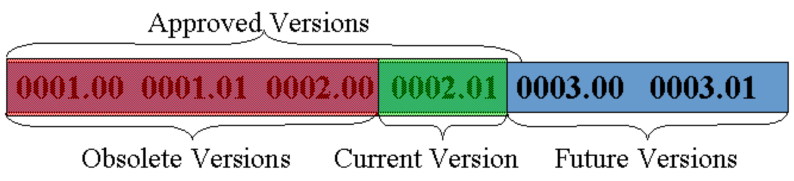
The picture below shows the evolution of a single version of a configuration object.



The version from which the new major or minor version must be copied can be independent of the version currently in memory. The user can select any of the existing versions (of this object) to start from.

Creating a new lower minor version (e.g. creation of minor version 0001.02 based on minor version 0001.01) is not possible when a higher major version (e.g. version 0002.02) is already current.

The picture below shows an example of creating new minor and major versions.



- Create a new major version based upon
  - minor version 0001.01 => newly created major version = 0004.00
  - major version 0003.00 => newly created major version = 0004.00
- Create a new minor version based upon
  - minor version 0002.01 => newly created minor version = 0002.02
  - minor version 0001.00 or 0001.01 => newly created minor version 0001.02 cannot be created because a higher major version 0002.01 is already current.
- The (approved) status is completely independent of the fact which version is “current”.

Important

No new minor version can be created based on configuration objects with version number 0000.00. For these objects, only a major version 0001.00 can be created (this is done automatically, even if the option **New minor version** has been selected in the GUI).

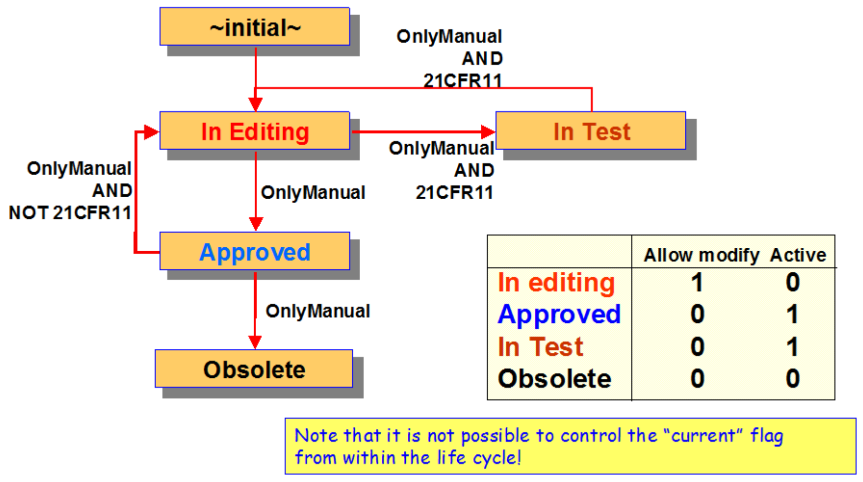
System lifecycle and version management

Version management dictates that, once a version is approved, it must never change. In such situations, the user must create a new version of the object. This implies that the status transition from **Approved** to **In editing** in the system life is not allowed.

For this reason, an extra check is added to the system life cycle for 21 CFR part 11 systems. A condition on the transition from **Approved** to **In Editing** checks if the user is working under 21 CFR part 11 mode or not.

For test purposes, the system life cycle has been extended in 21 CFR 11 mode with an **In test** state. This gives users the possibility to test their configuration, while still being able to modify the properties of a version. Objects 'In Test' are not current. As a consequence, if objects are assigned to the upper level with version '~Current~' (e.g. parameter profile to sample type), they will not be assigned on the operational level (or a previous, wrong version will be assigned). Therefore, it is important to configure fixed versions on the links. After testing, this can be updated if necessary.

The picture below shows the system life cycle.



	Allow modify	Active
In editing	1	0
Approved	0	1
In Test	0	1
Obsolete	0	0

Note that it is not possible to control the "current" flag from within the life cycle!

Important

In Unilab, the DBA is always authorized to execute any manual state transitions. This includes @A->@E system life cycle transition on a 21 CFR part 11 system. Since the **Modifiable** flag is switched on (by the transition to @E), the **Current** flag is switched off. This flag is no longer switched on when approving the object (hence, no current version of the object is available).

Please note that:

For GLP/GMP reasons, it is strongly advised not to perform daily Unilab operations, both configuration and operational, as DBA: other users – for example, Lims Administrator - are recommended in this case. This is especially true in a 21 CFR part 11 environment.

Should the situation described above occur, an E-mail will be sent to the responsible, set in the **DBA_EMAILADDRESS** system setting.

Current version

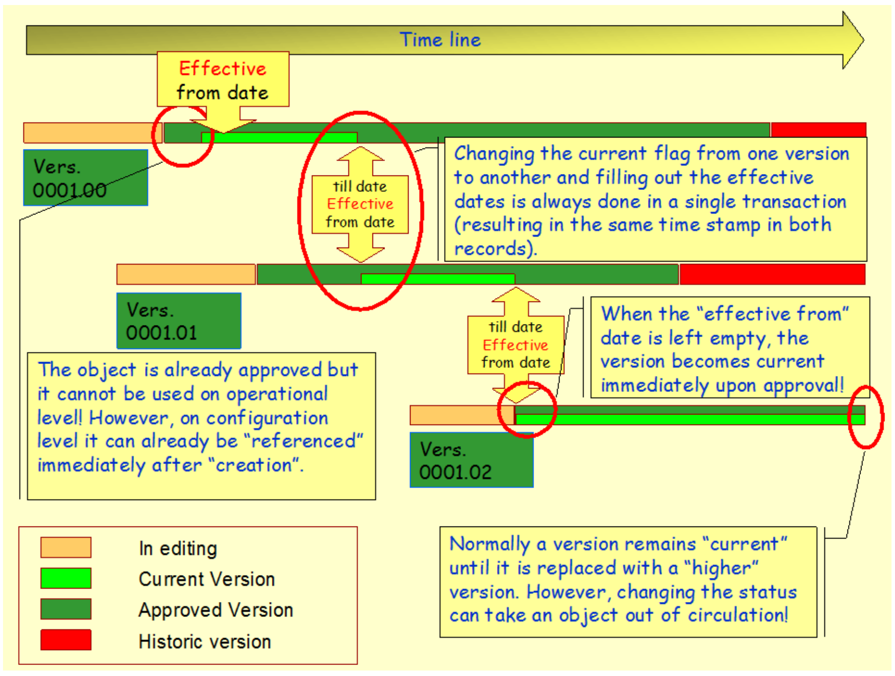
When defining a new version of a configuration object, the user can specify an **effective from** date to control when a specific version becomes the current version. The **effective till** date cannot be modified by the user (this field is always protected and is completely monitored by the system). The **effective till** date is filled in automatically as soon as a version is no longer current. This happens automatically when another object version becomes the current version. This means a version can become current only once.

The **effective from** date is just a standard property and can be modified on the property sheet at any time, assuming that the user is authorized to do this and the object is still modifiable.

Important

Once the **effective from** date is filled in, it can no longer be changed.

The picture below shows how to plan different versions of a configuration object.


Time line
Effective from date
Vers. 0001.00
till date Effective from date
Changing the current flag from one version to another and filling out the effective dates is always done in a single transaction (resulting in the same time stamp in both records).
Vers. 0001.01
till date Effective from date
When the "effective from" date is left empty, the version becomes current immediately upon approval!
The object is already approved but it cannot be used on operational level! However, on configuration level it can already be "referenced" immediately after "creation".
Vers. 0001.02
In editing
Current Version
Approved Version
Historic version
Normally a version remains "current" until it is replaced with a "higher" version. However, changing the status can take an object out of circulation!

A version can become current only if:

- The version is active.
- The version is not modifiable.
- The **Effective from** date is either blank or a date in the past.

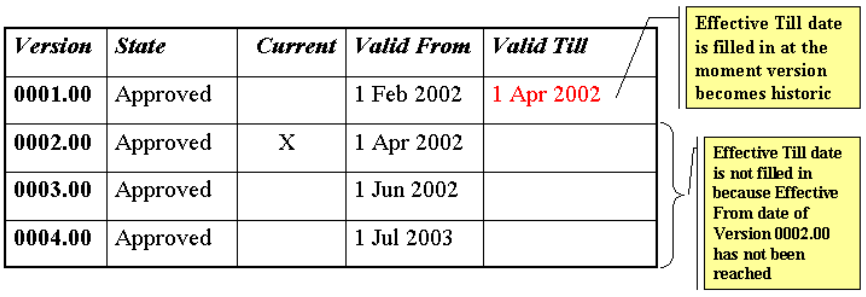
The following rules are applied to automatically monitor the effective dates (from and till) and the **current** flag:

- A specific **New Version Manager** job monitors all objects under version control. The default execution interval is daily (but can be customized via the system setting **NEWVERSIONMGR_INTERV**, similar to the timed Event Manager and/or the equipment intervention job).
- Whenever the user makes any changes to an object under version control, the following checks are performed:
  - If the **effective from** date is blank ,the **active** flag is set, the object is not modifiable and the status is not **In Test** (the status **In test** is hard-coded @T), then this version becomes current immediately (= within the same transaction of changing the status – within the Event Manager job)
  - If the **current** flag is set and the **active** flag is switched off or the **modifiable** flag is switched on, the **current** flag is switched off and E-mail is sent to the responsible to notify him about this very unlikely event! The e-mail address of this responsible is defined in the system setting **DBA_EMAILADDRESS**.

Important

The aforementioned situation may occur when executing the @A->@E system life cycle transition on a 21 CFR part 11 system as DBA. Please note that for GLP/GMP reasons, it is strongly advised not to perform daily Unilab operations, both configuration and operational, as DBA: other users – for example, Lims Administrator - are recommended in this case.. This is especially true in a 21 CFR part 11 environment.

The picture below shows how to plan new versions.



Version	State	Current	Valid From	Valid Till
0001.00	Approved		1 Feb 2002	1 Apr 2002
0002.00	Approved	X	1 Apr 2002	
0003.00	Approved		1 Jun 2002	
0004.00	Approved		1 Jul 2003	

Effective Till date is filled in at the moment version becomes historic

Effective Till date is not filled in because Effective From date of Version 0002.00 has not been reached

Assuming the current date is April 14th, and we're planning a version 0002.01 which should be valid from 1 May 2002:

• The Version 0002.01 becomes the current version on 1 May 2002

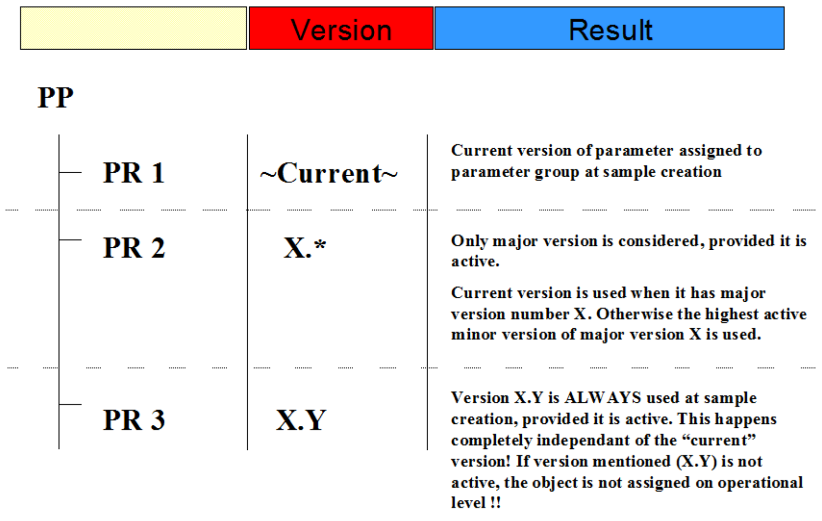
• The Version 0002.00 becomes historic on 1 May 2002

Handling a blank **effective from** date immediately allows introducing new minor versions instantly (because, in real life, it is sometimes necessary to take corrective actions – even in a strictly regulated environment).

12.5 Version Control on Used Objects

On the linked object level, the user can specify the version of the linked object (for instance, parameter profiles assigned to sample type).

By default, the version of the linked object is specified as **current**. This is interpreted as: always use the current version. The following figure provides an overview of all the possibilities.



PP	Version	Result
PR 1	~Current~	Current version of parameter assigned to parameter group at sample creation
PR 2	X.*	Only major version is considered, provided it is active. Current version is used when it has major version number X. Otherwise the highest active minor version of major version X is used.
PR 3	X.Y	Version X.Y is ALWAYS used at sample creation, provided it is active. This happens completely independant of the "current" version! If version mentioned (X.Y) is not active, the object is not assigned on operational level !!

Despite the fact that the current version of the used object is used as default, the version can still be overruled in the used objects property sheet. Any version of the used object can be assigned (e.g. a parameter-profile version that will become current in 3 months using a parameter and/or a method definition version that will also become current at that time).

12.6 Version Control on Operational Objects

Strictly speaking, there is no version control on the operational level! The system only maintains version information about the version of the configuration object that was used as a template to create the operational object.

Version indication on operational level

On the operational level, the version number that is displayed will be that of the configuration object on which the operational object is based.

Manual assignment of objects on operational level

When assigning objects on the operational level, the user gets a list to select from (by default, no version info will be provided). The system will now list all objects for which there is a current version.

12.7 Non-21 CFR 11 Systems

As described earlier in this chapter, version management is only available on systems with a SIMATIC IT Unilab Advanced Server license. On systems without this license, no version control is available.

This implies that on such systems:

- No version control is applied: changes to existing configuration objects do not trigger a new version. Obviously, these changes are logged in the object’s audit trail.
- The configuration object’s primary key on the database side also includes the object version. For non-21 CFR Part 11 systems, the version number is set to 0001.00. This version number is never updated/changed

If a new object is created, the **is_current** property is immediately set to **Current**. Whether or not the object can be used depends entirely on the settings for the object’s **Active** flag (determined by the object’s status).