Persistent States Concept

On this Page:

- States and Life Cycle
- Data Storage
- Signals

Related Pages:

- Getting Started with Persistent State Databases
- State Machine Diagrams
- Sending Persistent State Signals
- Persistent States of xUML Services on the Bridge

States and Life Cycle

Technically, creating a persistent state object means to create entries in generic database tables. In this context, the term generic means that the database tables do not depend on the object's class, because any <<PersistentState>> class will be mapped to the same tables.

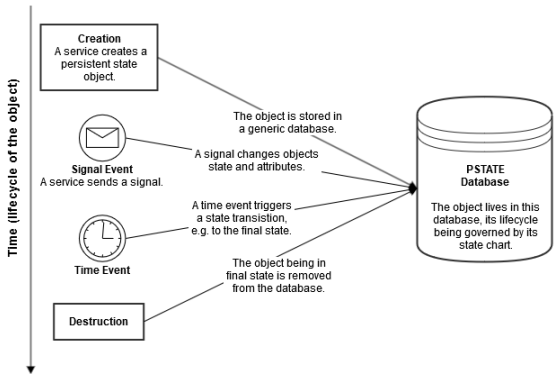
These tables hold information about the state of the object and all object data. After creation of a persistent state object, only the object's state machine diagram defines its life cycle. This means, only events defined in the state machine diagram can change object states and object data (for details see State Machine Diagrams).

Events in this context are:

- completion event: triggered by an automatic state transition
- signal event: triggered by a signal reception
- time event: triggered by a time trigger

For example, signals sent to the object transport data that can be added to the object. However, it is not possible to retrieve the object from the database, manipulate it, and write it back afterwards – though it is possible to retrieve a copy of a persisted object from the database. This means, after having created a persistent state object, the object lives on the database until it reaches its final state. Again, the object's state machine diagram defines which events may trigger the transition into its final state. If the object reached its final state, it will be deleted.

Figure: Conceptual Lifecycle of a Persistent State Object



This concept has several consequences:

- First, state charts describe the life cycle of objects completely. This means, the modeler is relieved from explicitly managing the destruction of persistent state objects.
- Additionally, state charts give a very concise and elegant overview of the life cycle of a persistent state object, thus increasing development speed and model maintainability.

Data Storage

The persisted data, current states and pending events are either held in memory, in a local file or an external RDBMS.

- **Memory** data storage has a huge performance advantage (5-15 times) over the others. The content of the state database is written to a local file upon shutdown and read from the same file when the service is started again. However, in the case of a power failure or server crash, any changes to the state database since the last start-up will be lost.
- A **local file** as data storage is about 5 times slower than memory. Unless the file system is corrupted, no data is lost upon power failure or server crash.
  As local file and memory use the same format, it is possible to switch between these configurations without losing any data. In both cases, the generated file **pstate.db** is a SQLite database file. For more information about SQLite and how to access this database, see Getting Started with Persistent State Databases.
- Using an **external RDBMS** is about 2-3 times slower than local files and requires some further actions by the modeler and database administrator to get started. For external data storage, the database dependency has to be drawn in the deployment diagram and an empty database schema/instance needs to be set up by the administrator. However, this enables the modeler to use any relational database being supported by the xUML Runtime.
  The major advantage of this approach is that backup, restore procedures and scaling mechanisms are delegated to the database management. Additionally any service that

accesses the same state database can send signals to the same objects. This includes, for example, services that implement load-balancing behavior sharing the same objects.
- A fourth option called **volatile** is similar to memory but does not backup/restore persistent state data to a file upon shutdown and restart. This storage method is recommended if the state database should be reset after each service restart.

It is not necessary to adapt database schemas when changing the structure of the persisted data because the database tables are generic. This makes the database model flexible and robust.

> How to operate xUML services containing Persistent State objects in various network setups is described on the Bridge User's Guide on Persistent States of xUML Services pp.

Signals

Sending signals means, that a signal message is queued in a table of the state database. Asynchronously, the xUML Runtime tries to deliver signals to its target object. The number of delivery retrials is limited and can be defined by the modeler (see Sending Signals). If the system does not succeed in delivering after the given number of retries or the object is destroyed before delivery, the xUML Runtime calls the optional default handler for the signal – if the modeler did define such a handler (see Handling Undeliverable Signals). Afterwards the undelivered signal is automatically removed from the queue and database.

In order to avoid concurrency issues, signal delivering and handling does comply with the following rules:

Rule	Description
Signals are never lost.	Every signal will be delivered to the object or external entity, to which it is directed, or after expiration will be handled by the default handler.
A signal is "used up" when it is accepted by an object.	The xUML Runtime removes the signal from the signal queue. This implies that signals cannot be consumed multiple times.
Signals are asynchronous.	This means that signals are not delivered "immediately" but some time after a signal is generated. In practice, this is most probably instantly, but may be delayed by concurrency, shutdown or other events.
Signals are being queued.	Multiple signals can be outstanding for a given persistent state object.
New signals are processed only after completion of the previous signal (Run-To-Completion).	When a persistent state object completes an event handler, it is in the new state. Only after completion of the event handler, the object can accept a new available signal if any such exists. This is called Run-To-Completion.
Sending order is preserved in one object context.	If a single object generates multiple signals to a receiving instance, the signals will be received in the order generated.
Sending order is not preserved over different object contexts.	If there are signals outstanding for a particular persistent state object that were generated by different senders, it is indeterminate, which signal will be accepted first. The signals are accepted in the order they are received in the queue, which may differ from the order they have been generated.