Basis Modeling

Technical
Documentation

Advanced Planner and Optimizer (APO)

By the Department
Basis Modeling
For internal use only

SAP AG
Neurottstraße 16
D-69190 Walldorf
Germany

50028362

December 1998

BME044.01

Table Of Contents

Table Of Figures

The following figures can be found at the end of the document:

Preface

Audience

This Bluebook introduces the internal structure of the Advanced Planner and Optimizer (APO). It concentrates on the core application that is strongly related to the liveCache technology. Application programers involved in the APO development will find an overview of the data model in conjunction with a description of the core functions. Apart from domain specific information, the design of the APO is described here as an example of how the liveCache technology can be used. This will be useful for developers of future liveCache applications.

Contents

The description in this Bluebook is based on APO release 1.0 that uses the liveCache release 7.1. In certain points, it gives a preview of features of future releases.

The first chapter introduces the basic terms of the APO. It gives a functional overview and describes the relations of the APO to other software systems.

Chapter 2 explains the architecture of the APO. After introducing all software components, a deeper insight of the liveCache layer is provided. This chapter also deals with the data model together with the core functions.

Chapter 3 examines the use of the liveCache technology in the APO. It starts with a brief introduction to the liveCache. Further subjects are object oriented design patterns and transactions.

Further Literature

For a deeper understanding of the liveCache technology please refer to the separate Bluebook 'liveCache'. The publications by the Basis Modeling Group as well as further material about the R/3 Basis System and related topics can be found at the following URL:
http://ency.wdf.sap-ag.de:1080/

A list of Bluebooks already published is available at:
http://ency.wdf.sap-ag.de:1080/bluebooks.htm

1. Introduction

This chapter introduces the APO basic terminology. It provides an overview of the application components responsible for supply chain planning. Furthermore, the relations of the APO to external systems will be explained.

1.1 APO Basic Terms

APS

APO

Advanced Planning and Scheduling (APS) is a category of business software. It is characterized by fast planning algorithms to be used with a large amount of planning data. To allow the execution of planning and scheduling functions in real time, the entire planning data is held in main memory. The APS software by SAP is called Advanced Planner and Optimizer (APO). It is implemented using an R/3 Basis System and the liveCache as platforms. **Figure 1** gives a functional view of the APO and related systems.

Application Modules

The main objective of APS is to support the generation of optimized executable plans in response to the rapid changes in supply or demand. The APO provides application modules for different tasks of supply chain planning: demand planning, supply network planning, production planning / detailed scheduling, and available to promise. The tasks differ in the following characteristics:

- location (whole supply chain or selected locations)
- time (long-, medium- or short-term planning)
- level of detail, determined by the considered order types

Supply Network

Most of the planning modules work on common data structures known as **supply network**. While the planning modules support the execution of planning tasks in real time, optimization tasks are processed by separate modules in batch mode.

APO System

All software components that work together to enable the planning and scheduling functions are regarded as an **APO system** or APO server. However, it is not a standalone component. An APO system depends on the cooperation with other systems like online transaction processing (OLTP) systems, e.g. R/3, and online analytical processing (OLAP) systems, e.g. the Business Information Warehouse (BW), to be supplied with data. These systems are responsible for maintenance and persistent storage of data as well as the execution of the plans made by the APO.

The following section describes the functional structure of the APO in conjunction with interfaces and possible relations to the outside. Throughout this Bluebook, the term APO will be used as a synonym with APO system.

1.2 Application Modules and Interfaces

As mentioned in the previous section, the APO provides several application modules for different tasks of supply chain planning.

Planning Modules

Production planning and detailed scheduling as well as supply network planning are based on a common data model, the supply network. Supply network planning considers the whole network. In contrast, production planning and detailed scheduling is limited to a selected set of locations.

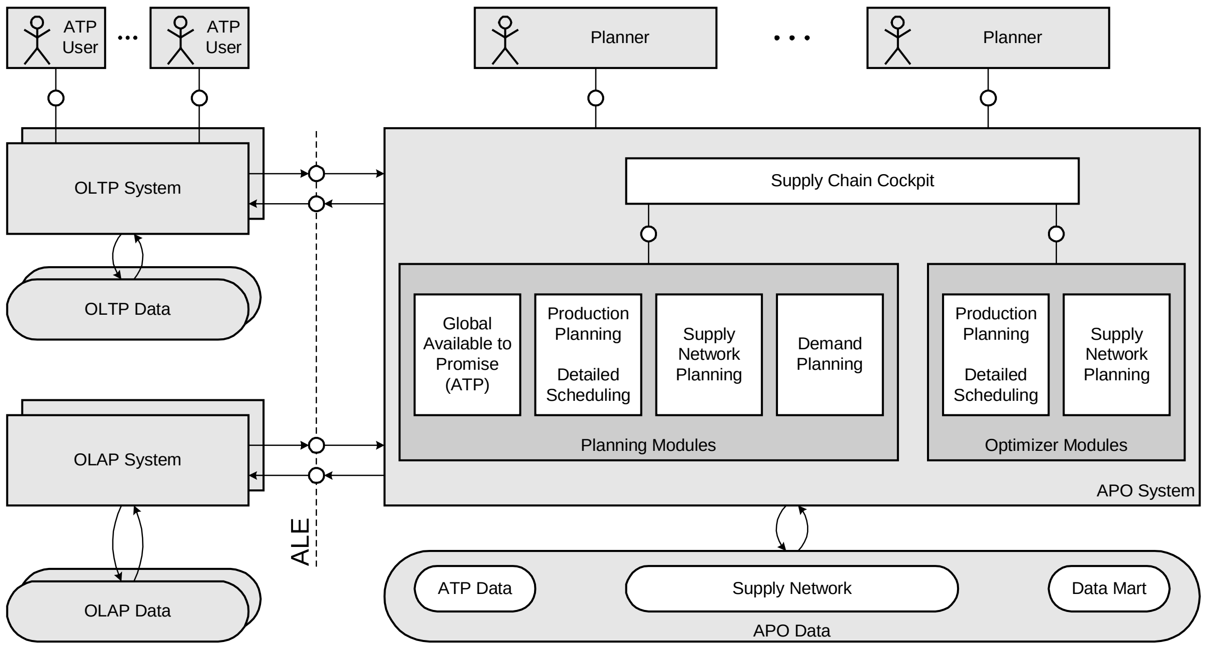


Figure 1 Modules and Interfaces of the APO

The level of detail increases from supply network planning to production planning and detailed scheduling. The **supply network planning** for instance concerns production, transport and storage capacities of larger units of planning. **Production planning and detailed scheduling** is based on capacities of single machines, labor pools and shift plans. According to their planning horizon, the planning and scheduling tasks can be classified as follows:

- short-term planning, e.g. detailed scheduling
- medium-term planning, e.g. production planning, supply network planning
- long-term planning, e.g. demand planning

The module for **demand planning** uses a separate data structure denoted as data mart in Figure 1. The **data mart** contains information like key performance indicators and supply chain costs. Data from the supply network is imported into the data mart, e.g. information about feasibility and profitability of plans. Forecasted demand is exported from the data mart into the supply network.

Another module relying on its own data model is **available to promise (ATP)**. The **ATP data** in turn is provided with information from the supply network.

Optimizer Modules

The algorithms for optimizing the supply network are different from those for online planning and scheduling. Usually, optimization requires some time and thus is executed in batch mode. Therefore, the optimization algorithms are implemented in separate modules. At present, **optimizer modules** exist for production planning and detailed scheduling, and for supply network planning.

The interaction of the APO application modules is not shown in Figure 1 because it is outside the scope of this Bluebook. However, the picture shows the interfaces for user interaction, and the connections to other systems.

Supply Chain Cockpit

With the supply chain cockpit (SCC), the APO provides a graphical user interface for modeling, navigating and controlling the supply network. The SCC is an ABAP application that uses a number of controls like maps and trees to allow navigation without passing multiple screens. Apart from their integration in the supply chain cockpit, application modules provide separate user interfaces in the style of R/3 dialog programs.

Users

For using most of the APO functions, the user must be logged on directly to the system. Typically, the number of such APO users is small compared with an OLTP system. An exception is the availability check provided by the ATP module. This function is intended for the use from an OLTP system via remote function call (RFC). Such ATP users are expected in a much higher number than the actual planners.

OLTP System

Since the APO covers only planning functions, maintenance and execution functions are in the responsibility of external online transaction processing (OLTP) systems. Thus, frequent data exchange between the systems is expected. Any kind of data, i.e. customizing data, master and transaction data, is involved. The OLTP system is not necessarily an R/3 System. It is also possible to connect more than one OLTP system to the APO, for instance if business applications like FI, SD and MM are distributed among different systems.

OLAP System

Historical data, cost information and forecasts may be maintained externally by an online analytical processing (OLAP) system, e.g. the SAP Business Information Warehouse (BW). The OLAP system provides the demand-planning module and the data mart with the necessary information. However, the existence of an OLAP system is not required.

In future releases, scenarios with multiple APO systems will be possible. For the time being, major technical problems concerning the distribution of the APO data are unsolved.

Details about the use of ALE in the environment of the APO as well as new requirements will not be discussed in this Bluebook.

2. Architecture

After explaining the structure of the APO from a functional point of view in Section 1.2, this chapter is a more technical approach. Starting with an overview of the main components, a detailed description of the core application in the liveCache follows. Finally, a brief description of the ABAP part is given.

2.1 Overview

The internal structure of the APO is depicted in **Figure 2**.

Three Tier Architecture

The APO is implemented using an R/3 Basis System. Thus, the architecture shows the typical three tiers:

- the (relational) database,[1] accompanied by the database management system which is responsible for the persistent storage of the application data.
- the application servers processing the actual application programs written in ABAP. The application servers and the database may run on different machines.
- the user interface clients (SAPGUI) running on the front-end.

This architecture is extended in some points. The **liveCache** provides an alternative storage at the database tier capable to store and process object data. The liveCache data is held in the main memory mostly. The **external servers** implement selected tasks with high performance demands, such as optimization. The SAPGUI is extended by custom controls for the Supply Chain Cockpit (SCC) not shown in Figure 2.

The APO specific parts are highlighted in dark gray on Figure 2. The application logic is distributed among the application servers and the liveCache. Much of the APO application runs on the application servers together with the processing of user interactions. These functions are implemented in ABAP and deal with relational data. The ABAP part of the APO application is divided into modules that correspond to the planning modules in Figure 1.

Planning and scheduling functions that are common to all modules and deal with object data form the core application of the APO. As shown in Figure 2, it is implemented in the **APO application component**, which runs inside the liveCache.

[1] Usually only one database exists in an R/3 System.

Some performance critical tasks are running on neither an application server nor the liveCache. They are implemented in external servers instead. Reasons for this are:

- The tasks do not rely on liveCache data.
- The tasks are expected to consume a large amount of processor time.
- Components not developed at SAP have to be integrated.

An example are optimizing tasks. They rely on the object data in the liveCache as well but normally cause high and long-time processor load.

All external servers and the APO application component within the liveCache are implemented in C++.

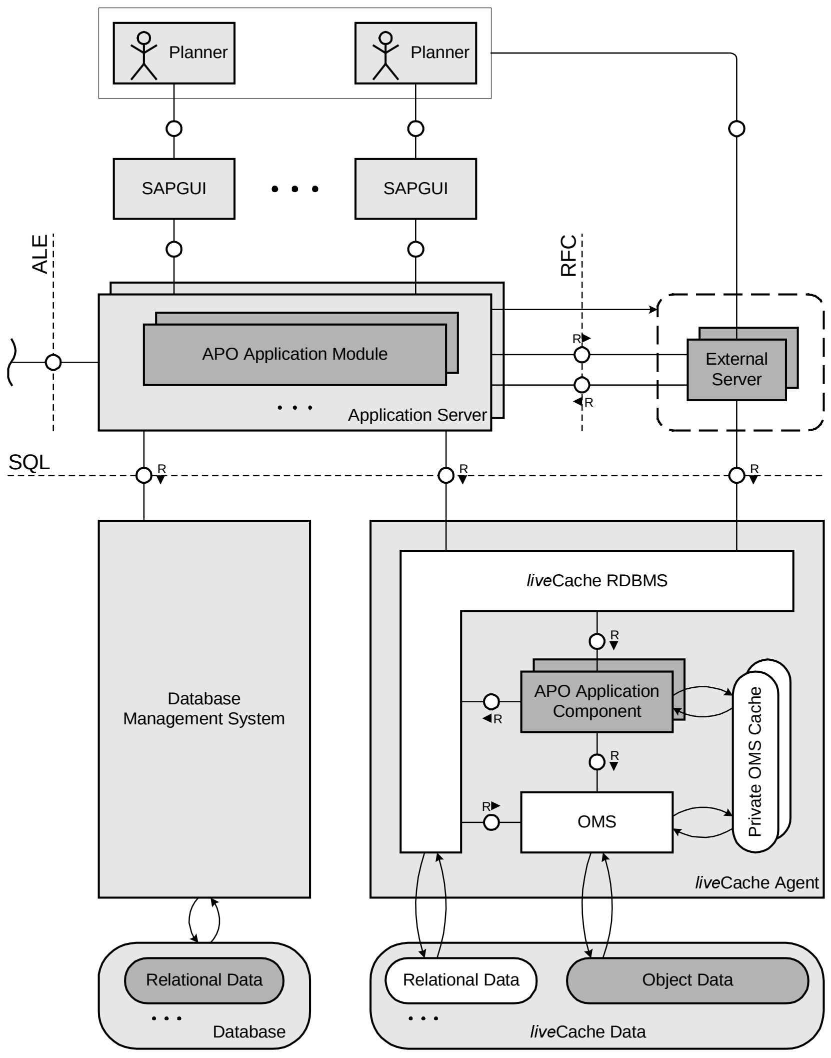


Figure 2 APO Architecture

liveCache

The liveCache brings application logic and data together in one large memory area. Furthermore, it provides an object oriented data model suitable for dynamically interlinked data structures. The application specific code is linked into the liveCache as a so called application component.[2] The liveCache relational database management system (RDBMS) is responsible for access and management of relational data. In addition, it handles the incoming requests from the application servers and external servers for calling methods provided by the application component. All requests to the liveCache have to be formulated in SQL.

OMS

Access and management of object data in the liveCache is performed by the **object management system (OMS)**. It supports concurrent access by locking and transaction mechanisms. Another feature of the liveCache is the consistent view during transactions. A description of the consistent view and the purpose of the private OMS cache can be found in Section 3.1. Object data cannot be accessed from outside the liveCache and thus is only manipulated by the APO application component. For a deeper understanding of the liveCache and its integration into an R/3 System, please refer to the Bluebook ‘liveCache’.

The decision of how to partition the functions between the liveCache and the application servers is influenced by the need of user interaction[3] and the distribution of data between the standard R/3 database and the liveCache. The following issues have been taken into account during the APO development:

- All data relevant for performance critical tasks should be stored in the liveCache.
- The size of each object in the liveCache should be kept small because a large number of objects is expected.
- The amount of relational data in the liveCache should be reduced in order to save space for object data.
- The number of types in the object data is restricted to 256 per liveCache instance.
- The relational data in the liveCache currently can only be accessed via Native SQL from within an ABAP program.

Object Data

The performance relevant data for the APO scheduling functions is stored in the liveCache using its object oriented data model. In particular, the order network as a part of the supply network, and the ATP data can be found here. The structures are designed to meet the needs of the different APO application modules. Planning and scheduling functions concerning these structures are implemented as methods of an application component running directly in the liveCache. For each session between an R/3 work process and the liveCache,[4] exactly one instance of the application component is created, accompanied by its private OMS cache. Objects that may be stored in the liveCache data are called **persistent objects**. Both, the data model of the core application and the structures inside the APO application component

[2] Technically, an application component is a COM object.

[3] The application component has no access to the user interface.

[4] In future for each SAP LUW.

will be described in Section 2.3. More technical information about the liveCache and its object model can be found in Section 3.1.

Relational Data

Forecast data used for demand planning as well as descriptive data not necessary for the core planning functions are stored as relational structures in the standard R/3 database. This was done in order to save space in the liveCache but also because ABAP Dictionary does not support the liveCache yet. Thus, the liveCache capabilities as relational database are currently unused in the APO. Section 2.4 provides information of how the relational data is structured and how it is linked to the object data.

External Servers

Some performance critical tasks of the APO application modules are implemented as separate servers. At the same time, the use of third party modules, not written in ABAP, providing highly specialized algorithms is possible. External servers exist for the following modules: detailed-scheduling optimizer, supply-network-planning optimizer, and forecast. The servers are pure C++ applications that provide an RFC interface.

Instances of some external servers live only for a single request, e.g. the multilinear regression used by the demand-planning module, and therefore need no further communication with the application module. Others perform callbacks to the application module in order to fetch the required data. The server for detailed-scheduling optimization even carries its own user interface written in C++. However, it may be migrated to the application servers and the SAPGUI later.

Generally, external servers are supplied with the appropriate data by the application modules. The data originates from either the relational or the object data. The detailed-scheduling optimizer is an exception and communicates directly with the liveCache via a separate connection.

Distribution

The APO makes no restrictions on the possible distribution of its components compared to an R/3 Basis System. In addition, the liveCache and the external servers may reside on separate hardware. There is a danger, that the liveCache may become a bottleneck in a large installation. Problems of partitioning or replication of data between multiple liveCache instances are not solved yet. Nevertheless, parallel processing is utilized on a multi-processor machine.

Software Layers

In the following sections, the APO application modules located on the application server together with the relational data structures will be called **ABAP layer**. The **liveCache layer** is represented by the APO application component in conjunction with the object data located in the liveCache. The optimizer and external servers will not be described further. The model of the liveCache layer will be refined in two ways: first by examination of the data structures and second by focussing on the functional elements that realize the core planning and scheduling functions.

2.2 liveCache Layer - Data Structure

The separation of data and function in the architecture of the liveCache layer is quite strict. The functions implemented inside the object data are restricted to the management of links to other objects. Planning functions are exclusively implemented in separate handlers that reside in the APO application component, on the other hand. A nearly complete picture of the data structures and handlers is given in **Figure A1** that can be found at the end of the document.

The object oriented data model introduced here is specific for the purpose of the APO. Other applications that make use of the liveCache technology will introduce their own models. Care must be taken for consistency and compatibility between data models of different liveCache applications.

Starting with an introduction, the following subsections describe the types of objects in the APO data model and the relations among them. Using liveCache terms, all objects introduced in this section are persistent objects. General information about the liveCache data model is given in Chapter 3.

2.2.1 Introduction

The master data necessary for planning and scheduling functions, together with the global planning state, are stored within the liveCache. They are characterized by a strongly interlinked structure. For an efficient mapping, the liveCache object model is used. However, the focus is mainly on keeping and providing information and mapping structural relationships. The actual scheduling functions are implemented in separate handlers. These are nonpersistent objects that form the main part of the APO application component. Handlers are described in Section 2.3.1.

The structures introduced in the following text are used to model the order network. The term **order network** denotes the part of the supply network made up of orders and their relations. The functions for supply network planning, production planning and detailed scheduling are based on these structures. The ATP uses its own data model that will be discussed in Section 2.2.4. However, the ATP data strongly depends on the order network.

Order

A simplified view of the main types and their relationships is shown in **Figure 3**. The topmost structure in the liveCache layer is an **order**. Orders may consume and/or produce a certain amount of material. The process of producing material may be divided into different operations each requiring resources, like people or machines, for a certain time. Many different structures of the ABAP layer can be modeled using a liveCache order, for instance: sales order, production order, purchase order, transport order, purchase requisition,[5] planned order, stock, or reservation. With the different order types of the ABAP layer mapped to liveCache orders, their

[5] In common SAP terminology ‘BANF’.

attributes and relations may have different meanings or may be subject to special restrictions.

Example

The following example illustrates the usage of the data structures throughout this section. Considering an assembled product in discrete manufacturing, not all features of the data structures can be examined. However, the example gives a rough idea of their purpose. The assembly of a certain amount of products is described by a production order. Components needed for the assembly are regarded as consumed material. The product itself is regarded as the produced material. An order has also relationships to other orders, e.g. the components are delivered from other production orders and the final product is assigned to sales orders.

The process of assembling the product may consist of several operations. The scheduling of a production order refers to the assignment of all operations necessary for assembling the product to concrete resources, e.g. machines or people. By allocating a certain capacity of a resource, the start and end times for the operations are defined. The scheduling algorithm simultaneously considers of temporal relationships between the operations, e.g. a given sequence, the capacity of resources, and the availability of material from related orders.

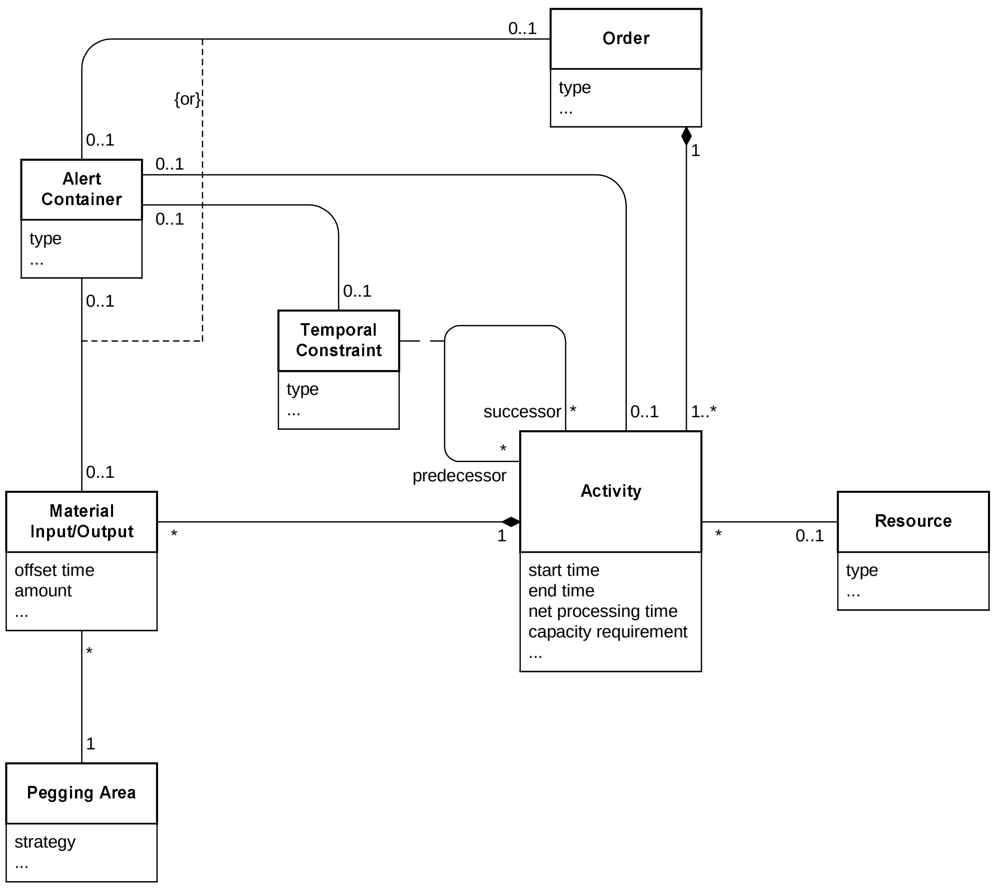


Figure 3 Core Data Structure (simplified)

Activity

In the liveCache layer, the representation or subdivision of an operation is called **activity**. For instance, an operation can be divided into activities for setting up, processing and tearing down. An activity itself is indivisible.

Every activity belongs to exactly one order. Like operations, an activity may be allocated to a resource. Attributes like start and end time, the net processing time, and the capacity requirement are regarded as planning data. They may be changed by the scheduling functions, e.g. by reallocating the activity to another resource.

Material Input/Output

A **material input/output**[6] represents either the consumption or the production of a particular material. A material input/output belongs to an order. However, it is solely connected to an activity that defines the time of material consumption/production. An attribute ‘offset time’ allows shifting that time relatively to the times of the related activity. The direction of material flow, input or output, depends on the sign of the ‘amount’ attribute.

The production order from the example above would have a material input for each component needed for the assembly. The product itself is modeled using a material output connected to the final activity (operation). Each material input is connected to the activity (operation) that processes the actual component.

Resource

The unified term for machines, people, production lines and groups of employees in the APO is **resource**. Resources are characterized by attributes varying over time. The latter are information about breaks, shifts, and workdays, as well as quantity-based capacity, e.g. the loading capacity of a truck or the volume of a tank. The system uses this data to calculate not only the available capacity (time- and quantity based) but also the actual execution times for the activities. Currently, resources allow at most one activity to be allocated at a time.

A general model for storing values varying in time in persistent objects, such as the capacity of a resource, is introduced in Section 2.2.2. A detailed view of resources, especially of their attributes and relationships to activities, is given in Section 2.2.3.

Pegging Area

The algorithm that establishes the material-flow relationships between orders, more concrete between material outputs and inputs, is called **pegging**. A material-flow relationship between two orders is called **pegging edge**. In the above scenario, pegging edges represent the supply of the assembly with components and the assignment of products to sales orders. Normally, such relationships are highly volatile. For instance, a component like a screw may be delivered from frequently changing suppliers. Thus, pegging edges have no direct representation in the order network. The data necessary for the pegging algorithm is stored in **pegging areas** instead. In a pegging area, all orders (material inputs/outputs) of a material are grouped together. A more precise view of pegging areas is given in Section 2.2.4 in conjunction with the ATP data model.

Temporal Constraint

Apart from the material-flow relationships, temporal relationships exist in the order network as well. They are expressed by **temporal constraints**. A temporal constraint describes the interval between two activities. Usually, temporal constraints exist between activities belonging to the same order.

[6] Called IoNodes in the APO code.

Each activity may have any number of successor and predecessor activities. In contrast to pegging edges, temporal constraints are fixed during the scheduling process. A temporal constraint is specified by:

- the type of constraint: minimal, maximal, minimal and maximal, and fixed interval length
- the reference points of activities: start-start, start-end, end-start, end-end

The reason for introducing separate objects for temporal constraints is the bidirectional character of the relationship. The decision was also driven by the mapping of the graph formed by temporal constraints and activities to the liveCache object model.[7]

Alert Container

It is unlikely that each capacity and material requirement or temporal constraint will be fulfilled during a planning session. Any violation of these conditions causes an **alert**. It provides information about the kind of violation and the affected objects. Alerts can be associated with material inputs/outputs, activities, temporal constraints and orders. Alerts about capacity overloads are associated to activities but also concern the allocated resource. The associated object is marked by a flag for the existence of alerts. An alert may indirectly affect more than one persistent object, e.g. an alert associated with a constraint affects adjacent activities. Such dependencies need not be stored in the data model because they are known in advance and do not change.

The **alert container** stores all alerts related to a single object. It holds a compressed representation of the alerts, not visible to the outside. The management of alerts and alert containers is done by a separate handler that prepares the information according to the needs of the ABAP layer.

Future Releases

Some scenarios could not be mapped appropriately to the data model described above. Thus, it has to be extended or changed in future releases. The following features are expected for APO release 1.1. The list does not claim to be complete, however.

- Activities may be allocated to several resources simultaneously.
- Subsequent activities, e.g. an operation that consists of setting up, processing and tearing down, can be allocated to a resource without interruption by other activities.
- Setup matrices allow the calculation of times for setup activities according to the preceding operation.
- For modeling groups of machines or employees, more than one activity can be allocated to a resource at the same time.
- Apart from exact times, scheduling can be based on larger units of time, for instance days or weeks. This is also known as **bucket-oriented scheduling**.

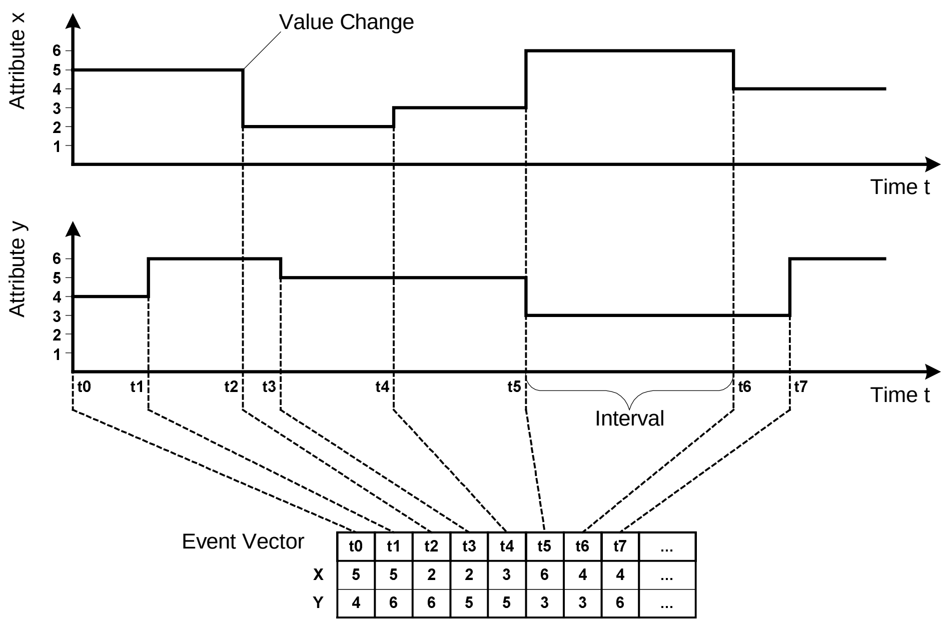
[7] See Section 3.2.1 for more information on the technical realization of interlinked structures.

- In addition to **dynamic pegging edges** that may be changed by the pegging algorithm, pegging areas store **fixed pegging edges** that can only be changed manually.
- The distinction of pegging areas by locations and batches is possible.
- Custom characteristics can be assigned to material inputs/outputs, for instance the quality of the same type of screws, and are distinguished by the pegging algorithm.

2.2.2 Event Vectors

The concept of **event vectors** is frequently used within the APO for attributes; their values have to be stored for several points in time. Examples are time lines, period splits, or the ATP time axis. These will be introduced in the following section.

Every change of an attribute value determines exactly one entry in the event vector. An entry covers the time of the value change together with the current attribute values. The values are valid up to the next entry. The intervals between two entries may be of arbitrary length. **Figure 4** shows an example where the event vector is determined by two attributes.



	t0	t1	t2	t3	t4	t5	t6	t7	...
X	5	5	2	2	3	6	4	4	...
Y	4	6	6	5	5	3	3	6	...

Figure 4 Event Vector

Event vectors provide a compact representation of nonperiodically changing values and allow binary search for a desired interval. A criterion for combining several attributes in an event vector is that they are frequently accessed together.

2.2.3 Resource and Activity

The general attributes of resources and their relationships to activities have already been described in a previous section. In fact, resources are implemented in more than one object type as shown in **Figure 5**. A resource references a time line. Regarding of bucket-oriented scheduling, it also references a period split and consists of a bucket vector. A list of alternative resources, optionally owned by activities, references one or more resources.

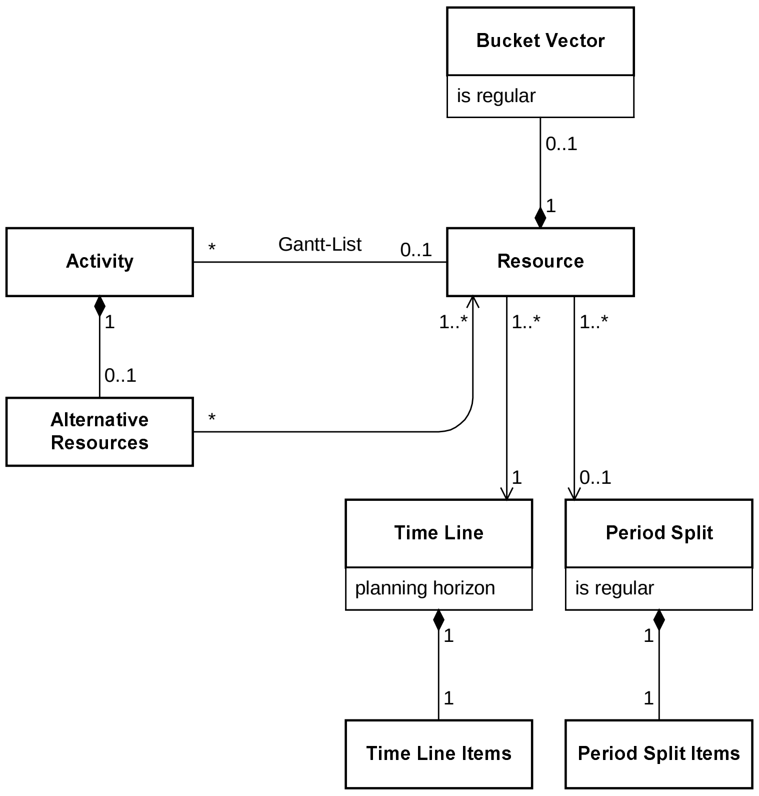


Figure 5 Resource and Activity

An activity is allocated to at most one resource at a time. All activities allocated to a resource are maintained in a 'Gantt-List' according to their start times. In fact, there is a second list, not shown in Figure 5, for temporary deallocated activities. Activities that do not depend on a special resource may have a list of alternative resources. This list contains references to resources the activity could be allocated to instead of the primary resource. Planning attributes of the activities are stored with each reference.

In conjunction with the activities, the concept of alternative resources experiences a complete redesign in APO release 1.1.

Time Line

Attributes of resources that are variable over time are stored in **time lines**. Examples are the time-based capacity, the quantity-based capacity, the rate of resource utilization, planned breaks, and shifts. Scheduling of activities using exact times is based on these attributes. Time lines (or more concrete: time line items) use the model of event vectors. The validity of a time line is limited by the 'planning horizon' attribute. Sharing time lines between multiple resources has to be done with caution because they contain resource specific information as the available capacity.

Bucket Vector

Apart from scheduling for exact times, the APO release 1.1 allows bucket-oriented scheduling for larger units of time. Resources that consist of **bucket vectors** are called bucket resources. A bucket vector provides an aggregated view of the attributes in the time line. It is a sequence of **buckets** each specified by a time interval, for instance a day or a shift, and the aggregated values from the time line for that interval.

The unit of the bucket vector, days, weeks, or shifts, defines the length (of time) of each bucket and is the same for all buckets. Nevertheless, the length of each bucket may differ in case of shift-based units. Shift-based units are called **irregular,** while **regular** units are based on days or weeks. The unit of a bucket vector determines the smallest possible unit for scheduling the bucket resource.

Period Split

Period splits[8] allow varying the unit of time for bucket-oriented scheduling. Each entry in period split items determines the unit for a particular planning period. The units of all periods in a period split have to be either regular or irregular. Furthermore, the type of unit must be the same like in the bucket vector of the referencing resource. The period split items again are event vectors.

Since the unit of the bucket vector is the smallest possible unit for scheduling a bucket resource, all units in the referenced period split must be coarser. Period splits only contain information about the level of aggregation for different planning periods. The actual planning data is drawn from the bucket vector. The data has to be accumulated from the buckets in case of scheduling periods with coarser units.

Together with time lines and period splits, bucket vectors are filled up with data from the ABAP layer. Typically, the scope of the bucket vector will exceed the planning horizon of the time line belonging to the same resource.

Time lines and period splits can be shared between several resources and thus need to be maintained separately. The actual data is stored in the time line items respectively period split items, while time line and period split act as proxies to enable access outside the liveCache layer. This is necessary because variable length objects do not allow access via a hash key.[9] In sum, alternative resources, bucket vectors, time line items and period split items cover a variable amount of data.

2.2.4 Pegging Area and ATP Data

In the previous section, the structures necessary for scheduling orders/activities regardless of information about material flow have been

[8] Called ‘AggregationGridObject’ in the APO code.

[9] The different types of liveCache objects and their usage in the APO are described in Chapter 3.

described. This section deals with the structures necessary for pegging and ATP. Both have close relationship as depicted in **Figure 6** and thus are discussed here together.

Pegging Edge

A relationship representing the material flow is called pegging edge. It can be established between two activities and affects only one particular material. A pegging edge is characterized by the amount of material delivered from an output to an input. Each input and output may be connected by any number of edges. Mostly, pegging edges are regarded as highly volatile objects. They are called dynamic edges. A dynamic pegging edge has no direct representation by a persistent object. Instead, the information is stored in the pegging area. In some cases pegging edges are established manually by the planner and must not change during the execution of the pegging algorithm. Such edges are called fixed.

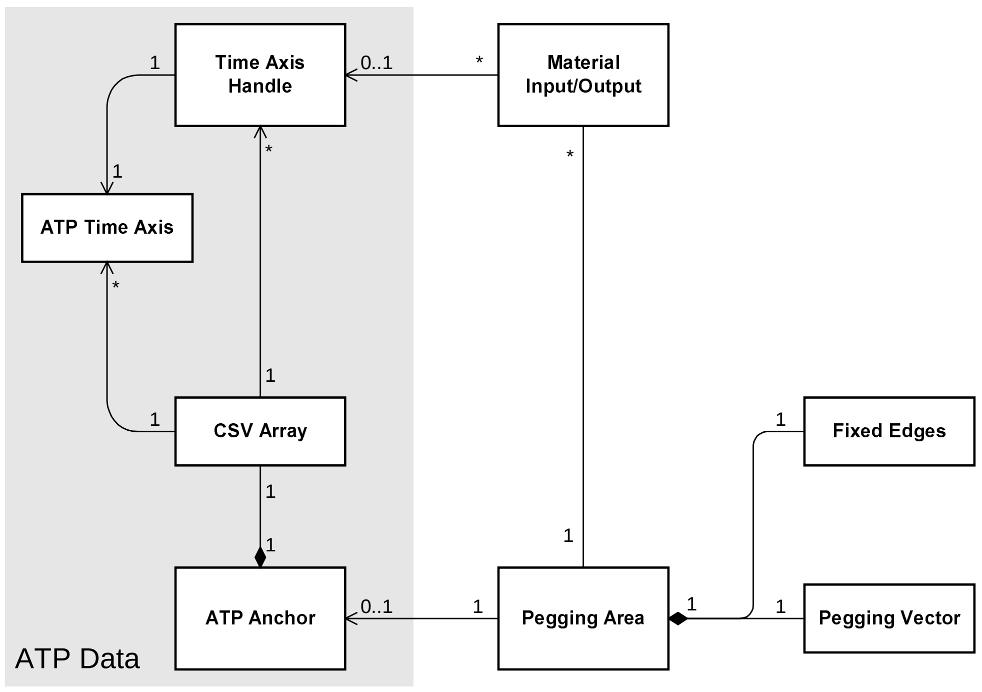


Figure 6 Pegging and ATP

Pegging Area

All inputs/outputs of a material in a material resource planning (MRP) area[10] are held in a pegging area. Pegging areas form disjunctive sets of material inputs/outputs. The material flow between pegging areas is outside the responsibility of the liveCache layer and may be carried out by the optimizer.

The state of the dynamic pegging is represented by the pegging vector[11] that is again an event vector. For each interval, the deficit and surplus as well as references to the first material input and output in the interval are stored. In some cases (make-to-order production), the dynamic pegging is not desired. Therefore, a set of fixed pegging edges is contained in a pegging area.

[10] I.e. storage locations or production lines of a plant.

[11] The so called ‘Stützstellen’ in the APO code.

ATP Data

The available to promise (ATP) module allows a fast check whether a material (product) is available for a customer and triggers its production if not. The data necessary for this check is drawn from the order network and stored in separate structures in order to increase the performance. Changes to material inputs/outputs and pegging areas that occur during scheduling have to be passed to that structures. The ATP module only reads from the ATP data. In contrast to the order network, persistent objects representing the ATP data cover much of the application logic, e.g. the ATP anchor.

ATP Time Axis

The data representing the availability of material is stored in the **ATP time axis**. An entry consists of a date, an open and a confirmed quantity. The ATP time axis is another example for an event vector.

ATP Anchor

The ATP time axis can be accessed via either the **ATP anchor** or the ATP time axis handle. An ATP anchor corresponds with a pegging area and thus is responsible for a single material in a location. It is the access point for the ABAP layer and also implements the core algorithm of the ATP check.

CSV Array

In order to allow a more detailed classification of material, an ATP anchor contains a **CSV array**. The abbreviation stands for category, sublocation and version. A row in the array is made up of the three attributes together with references to an ATP time axis and an ATP time axis handle. The ATP time axes only store information according to the classification in the CSV array.

ATP Time Axis Handle

The **ATP time axis handle** was introduced to speed up the access to ATP time axis from the order network. For that reason, material inputs/outputs are provided with a reference drawn from the CSV array. ATP time axis handles cover functions for inserting, deleting and changing entries of the associated ATP time axis. They are responsible for exactly one ATP time axis.

The APO runs without ATP functions, optionally. Thus, the ATP data structures need not to be used. This is expressed by the 0..1 cardinalities between material inputs/outputs and ATP time axis handle, and pegging area and ATP anchor.

2.3 liveCache Layer - Functional Structure

In the APO application component, two functional layers can be distinguished: the API and the handlers. **Figure 7** shows an instance of the APO application component. As depicted, all functional objects, such as API and handlers, exist only once per instance. The persistent objects, in contrast, are shared between all instances of the APO application component. Details about the design of handlers and the concurrent access to the persistent data can be found in Section 3.2.3.

Handler

A **handler** is a nonpersistent object with focus on behavior. Several approaches for grouping functions in a handler can be found in the APO application component. As shown in Figure 7, some of the handlers, such as net changer, scheduler, pegging handler, and slotter access data from different types of persistent objects. The functions of such handlers are defined by their domain. Others like the alert handler, time line handler, bucket handler and period split handler are responsible for one type of persistent objects. Except from creating or destroying, handlers of the second kind serve all requests concerning the corresponding persistent objects.

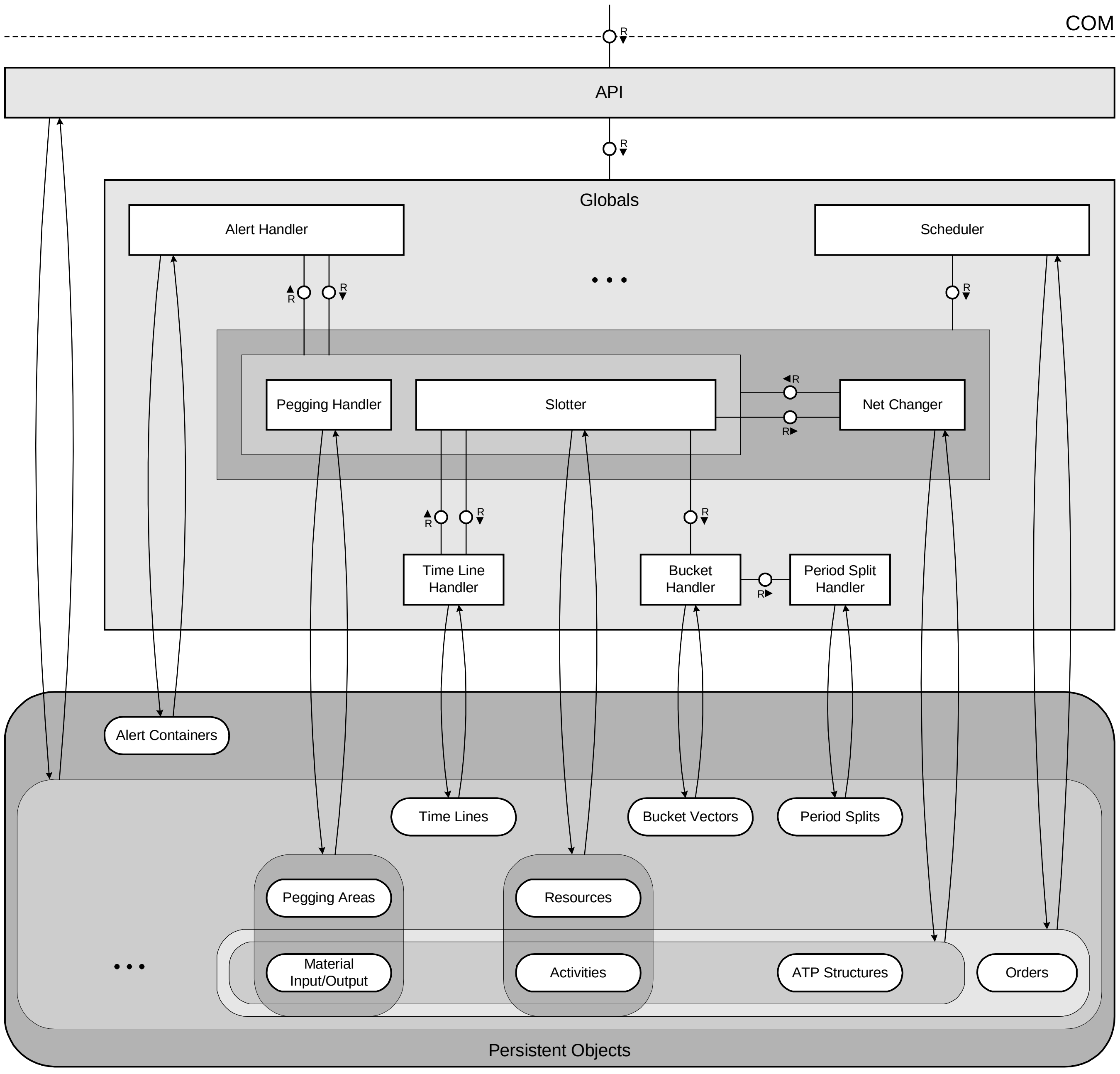


Figure 7 Handlers

Handlers have no inherent activity. They exclusively serve one request at a time. Every request to a handler has been originally triggered by the API but may be subdelegated to other handlers. All instances of handlers belonging to an APO application component are put together in an object called **globals** to ease the communication with the API.

API

The **API** provides all functions necessary for the ABAP layer. The functions are grouped into smaller packages according to related tasks. The packages adapt the functions offered by handlers to the different needs of the APO application modules as well as provide the functions to create, change and delete persistent objects. The subsection about the API provides detailed information about this topic.

As depicted in Figure 7, the API totally hides all handlers from clients; therefore, they are not visible from the outside. On the other hand, persistent objects are visible to the ABAP layer, though in different appearance or only by a GUID.[12]

2.3.1 Handlers

This section briefly describes the tasks of each handler, how it uses services of other handlers and which persistent objects it affects. These relationships are depicted in Figure 7 that shows the main handlers and persistent objects.

In order to reduce the number of communication channels and read/write accesses in the diagram, elements with common relationships are grouped together in thin lined boxes. These boxes are not containers like the globals but simplify the diagram.

Scheduler

The complete scheduling task is controlled by the **scheduler**. It makes extensive use of pegging handler and slotter, while all requests to change planning data are delegated to the net changer. The scheduler provides functions to schedule either a set of activities or a single order. Different options are offered like taking account of temporal constraints or using pegging.

Pegging Handler

The dynamic pegging and the management of fixed pegging edges is done by the **pegging handler**. It allows retrieving the internal state of a pegging area by a rich set of functions. The pegging handler can be regarded as the managing component of pegging areas and material inputs/outputs. Though modifications to pegging areas are performed by the pegging handler, the task is observed by the net changer. Thus, all requests from the scheduler for changing material inputs/outputs and pegging areas have to be sent to the net changer. The net changer in turn requests the pegging handler.

Slotter

The actual task of scheduling activities and allocating resources is done by the **slotter**. It employs a number of specialized handlers to accomplish this task. In case of scheduling using exact times the slotter requests the **time line handler** for information about available capacity or possible start times. The time line handler in turn notifies the slotter about changes at the time line. For bucket-oriented scheduling, the **bucket handler** is called to get the appropriate information about times and capacities. The bucket handler uses the **period split handler** in order to get the current unit of time for bucket-oriented scheduling.

[12] Globally Unique Identifier: used as unique name for persistent objects in the ABAP layer.

Net Changer

All requests to change attributes or links of persistent objects have to be sent to the **net changer**. A record of the request is kept there before the task is delegated to the appropriate handler, e.g. the slotter or pegging handler. Changing attributes of material inputs/outputs and activities is directly performed by the net changer.

In addition, the net changer provides functions to control the recording of changes to objects, and to replay the last recorded operations. Such behavior is useful e.g. for the simulation of successive operations.

Alert Handler

The **alert handler** has the sole responsibility for alert containers in the liveCache layer. These are completely hidden from the clients. On a client request, the alert handler reads the alerts from the alert container assigned to the specified object. Though the alerts have the characteristics of events, the alert handler is a passive component, which means the occurrence of an alert will not be recognized until polling from a client at the API.[13] The reason for this is the lack of a callback mechanism from liveCache to ABAP application modules.

Furthermore, the alert handler is notified about alerts detected by the slotter and the pegging handler during their tasks. It is also responsible for managing the alert containers.

2.3.2 API

The internal structure of the API is depicted in **Figure 8**. The packages and their communication with handlers are shown there in particular. To simplify reading the diagram, some handlers and API packages with common relationships are grouped together.

Packages

Packages are groups of API functions and, in contrast to handlers, they have no internal state. Packages distinguish internal functions only visible inside the API and interface functions accessible from the outside.

The core API provides interface functions for accessing the persistent objects as well as frequently used internal functions. The different order types in the ABAP applications are mapped to the unified order type in the liveCache by the production planning, transfer and purchase API. In addition, they provide the planning functions for the related applications. Purchase orders together with customer orders and stocks are handled by the purchase API.

Functions for retrieving a view of the planning data needed by the user interface are grouped in the planning table API. The optimizer API contains corresponding functions needed by the external optimizers. The remaining packages are specialized for the management of particular object types, e.g. time line, resource or period split, or they act as proxies for a handler like the alert handler.

[13] From the alert monitor, an ABAP application that calls the alert handler API.

As shown in Figure 8, the ATP package and the net checker package do not call any handler. Instead, they access the persistent objects directly.

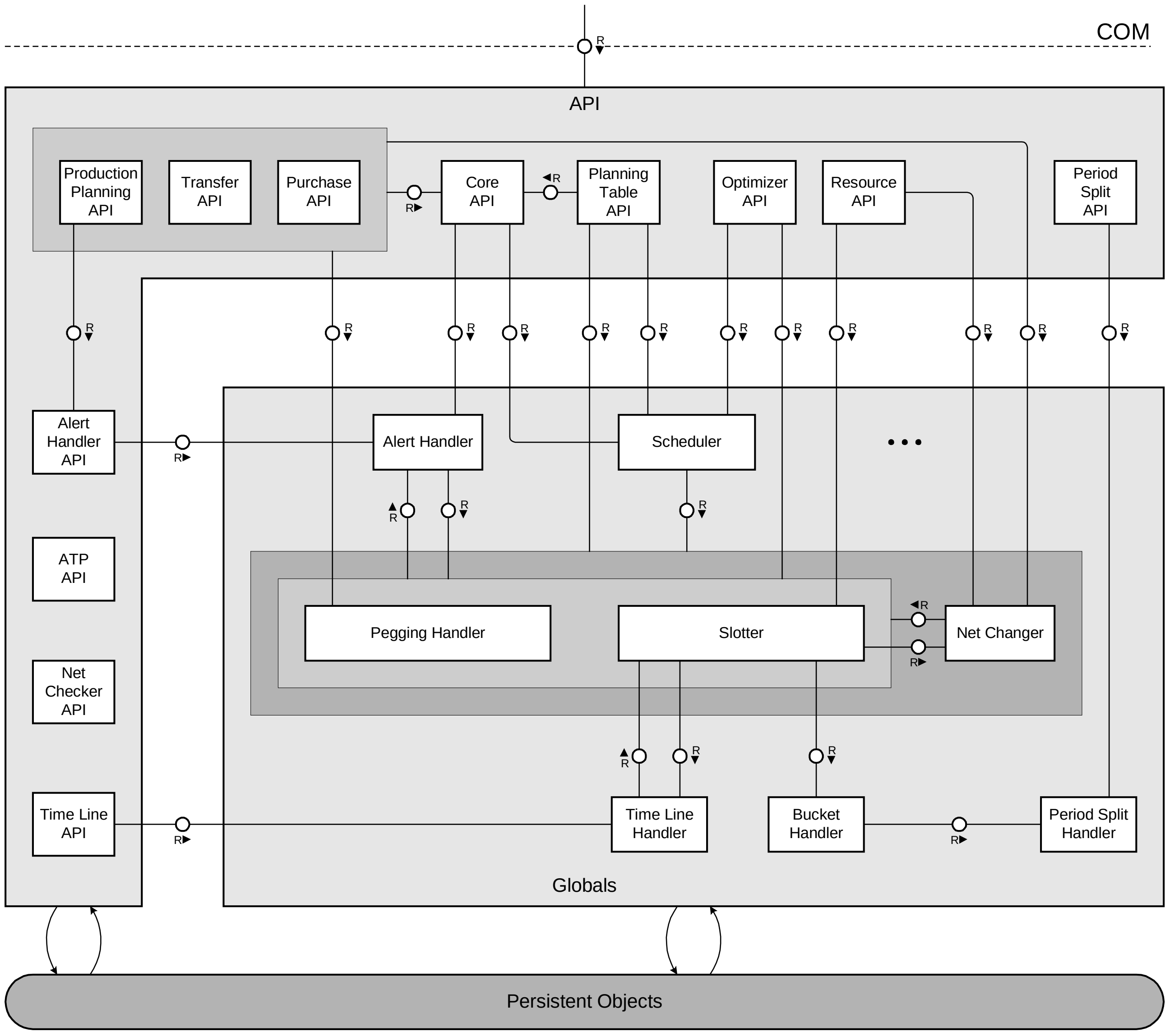


Figure 8 API Structure

Functions

API functions exist in two variants: for processing a single persistent object and for bulk processing. An example for API functions is the creation of orders where both variants exist.

Most API functions are characterized by the following tasks:

- parameter checking
- access and conversion of ABAP internal tables (call and return parameter)
- mapping of ABAP types to liveCache types and vice versa (ABAP structures to C++ objects, different time representation)
- transaction management

- exception handling

The transaction management is important in conjunction with exception handling in order to keep the consistent state of the network. While API functions processing a single object are atomic, bulk processing may succeed only for a subset of the addressed objects. Details about the application of the liveCache transaction concept to API functions can be found in Section 3.2.4.

2.4 ABAP Layer

This section introduces the design of the ABAP part of the APO. It is intended to raise one’s eyes out of the liveCache to complete the view of the APO. However, it does not claim to be a sufficient explanation, this is outside the scope of this Bluebook.

Data Structure

As mentioned in Section 2.1, only the performance relevant data of the supply network are located in the liveCache layer. Most of these data structures appear also in the ABAP layer, but in a different representation. Exceptions are activities that can be accessed from the ABAP layer, but no additional information is stored there. On the other hand, some structures of the supply network have no representation in the liveCache at all. For instance, locations or transport relations are completely implemented in the ABAP layer.

Anchor Table

The part of the supply network located in the ABAP layer is based on a relational data model. The structures are mapped to tables, and the relations between them are represented by foreign-key relationships. The connections to the order network in the liveCache are implemented using **anchor tables**. A special field containing the GUID[14] of the related liveCache object acts as a link into the order network. The technical background for connecting the relational and the object model can be found in Section 3.2.1. Apart from the GUID, an anchor table contains further information about the linked object, i.e. descriptive data not necessary for the core scheduling functions but useful for identifying or retrieving the object.

Data Mart

As described in the first chapter, demand planning is based on its own data structures completely different from the supply network. The union of all structures used for demand planning is denoted as data mart. Some structures are borrowed from the BW like InfoCubes and hierarchies. Consequently, the BW OLAP processor is included in the APO system.

Functional Structure

The ABAP layer of the APO is divided into the application modules introduced in Section 1.2 as well as a module implementing the supply chain cockpit.[15] Two additional software layers, the object and the data management layer, cover common operations of the application modules. They form a two-layer abstraction for accessing the supply network.

---

[14] Globally Unique Identifier.

[15] The namespace for the APO development is /SAPAPO/*.

The access to the liveCache object data is encapsulated in the object management layer. It is responsible for maintenance of connections with the liveCache as well as it covers details on calling methods of the APO application component like parameter passing or return code handling. The data management layer is based upon the object management layer and brings together the handling of relational data and object data.

3. Implementation

This chapter examines in detail how the liveCache technology is employed in the APO. It starts with a brief introduction of the liveCache and further concentrates on the APO specific usage. The section covers the features of liveCache release 7.1.

3.1 liveCache Architecture and Features

Database Interface

The architecture of the liveCache and its integration into the APO system has already been shown in Figure 2. In fact, the liveCache is a second, alternative database server. At present, according to the R/3 database interface design, the liveCache cannot be accessed via Open SQL from an ABAP application.

RDBMS and OMS

The liveCache is a hybrid database which consists of the relational database management system (RDBMS) and the object management system (OMS). Together with the liveCache data, all parts of the liveCache agent are held in main memory to reduce the response time for requests. The relational data can be accessed from the outside via SQL requests to the RDBMS. The object data is not directly accessible from the outside. Application specific code must be linked into the liveCache in order to access and process the object data. The liveCache can be extended by **application components** that run in the address space of the liveCache agent. An application component has to be implemented as COM object[16] provided by a DLL. The code in the application component can access relational and object data via interfaces to the liveCache RDBMS and the OMS. From an ABAP program, methods of the application component can be called using the SQL statement execute procedure that performs a synchronous procedure call. This is similar to the concept of stored procedures in other relational databases. Callbacks from the application inside the liveCache to the ABAP program are not possible.

Application Component

Persistent Object

The further descriptions concentrate on the OMS, because the APO currently uses only liveCache object data. In this area, a piece of data that can be read, locked and written is called **persistent object**. However, the term persistent is used here in the sense of a consistent state for multiple users that is guaranteed by the OMS rather than of durability and fail save storage. To store objects in the liveCache data, their classes have to be derived from either OmsObject or OmsKeyedObject.

OID

Each object is assigned a unique **object identifier (OID)** which is used for referencing it in the liveCache. However, an OID is not unique across time. This means an OID may be reused for a newly created object once the previously assigned object has been deleted. If derived from

[16] Currently only C++ and Windows NT on Intel platforms are supported.

OmsKeyedObject: Persistent objects alternatively can be addressed by a user defined key.

Before persistent objects can be accessed, the related OID has to be dereferenced. The OMS creates a copy of the object in the private OMS cache of the requesting application. Then the application is free to access and change this copy exclusively until the next commit or rollback. If the changes to the private copy are to become visible to other applications, an explicit store command is necessary.

Variable Length Object

Variable length objects are persistent objects that allow storing array-like structures that do not have a fixed length in the liveCache. A variable length object can be accessed using an OmsVarOid which is a special type of OIDs. When accessing an OmsVarOid, the entire object is affected, i.e. loaded, locked and stored.

liveCache Session

Each work process in an application server is connected to exactly one **liveCache session**. However, this session can be established and destroyed by the application programs running in the work process. After starting a new session, the appropriate application component is instantiated at the first call to one of its methods. Due to the dispatching of ABAP programs onto different work processes by the application servers, liveCache sessions may be shared by different ABAP program contexts. Consequently, the application components belonging to a session may be shared too. This affects the design of the application code because a method can never rely on the fact that subsequent calls belong to the same SAP-LUW. In conclusion, the internal state of an application component is invalid at least after the replacement of the current ABAP program in the work process which is possible between two dialog steps. To address this problem, the linkage between application component and session will be removed in the liveCache 7.2.

Transaction

Typically, a liveCache session is divided by commit or rollback into several **transactions** (DB-LUWs). A transaction again can contain several **subtransactions**. Transactions embrace changes to the relational and the object data.

Transaction Mode

Before an application method is allowed to store a persistent object, it has to acquire the lock for it. Three different transaction modes are offered by the liveCache.

- In the **implicit mode**, the object is locked automatically at the first dereferencing of its OID in a transaction.
- Locks have to be acquired explicitly in the **optimistic mode**. All requests for locking and storing objects are delayed up to the end of the current transaction.
- The APO release 1.0 uses the **pessimistic mode**. Locks also have to be acquired explicitly but all requests for locking and storing objects are executed immediately.

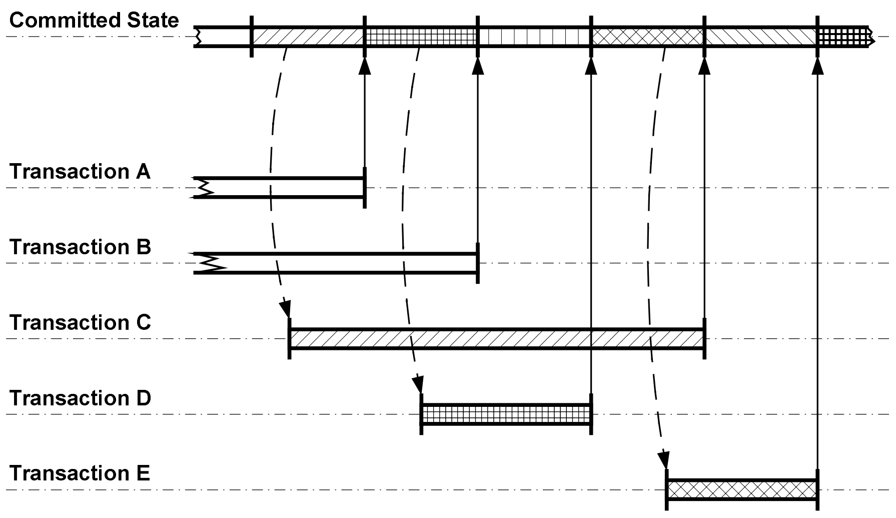


Figure 9 Consistent View of Multiple Transactions

Consistent View

The progress of the committed state of persistent data is driven by the changes each application commits at the end of a transaction. For each transaction, a **consistent view** is guaranteed by the liveCache. It can be regarded as a snapshot of the committed state at the beginning of the transaction. The consistent view lasts until the end of the transaction. A sample scenario is shown in **Figure 9**. The committed state changes with the end of each transaction denoted by a solid, vertical arrow. The consistent view of each transaction corresponds to the committed state at the time it was started depicted by a bent, dashed arrow. Every time a transaction dereferences an OID, the appropriate object will have the same state it had at the beginning of the consistent view.

Version

The consistent view is extended by the concept of **versions**. A version may be kept over several transactions. It is exclusively assigned to one transaction at a time. Versions may be stored and reloaded, which allows the maintenance of several versions for a single user. However, the changes made to a version can never be stored back into the committed state. In future releases of the APO, the concept of versions will be used to support simulated planning operations.

3.2 APO Specific Usage of the liveCache

Data Structure

The approach of the liveCache technology is to place the performance critical data together with the core functions in an object oriented, memory based environment. The liveCache part of the APO in turn is separated into data structure and functional structure. This is done to meet the needs of multi-user, real-time operations. The data structure described in Section 2.2 covers the user-independent, global state of the planning data. The structure is designed for a space saving storage and follows the limitations of persistent objects in the liveCache, e.g. the fixed size.

Functional Structure

All planning and scheduling functions are performed upon the persistent data. Most of the functions are implemented separately from the data, encapsulated in handlers residing in the APO application component. For each session, a single instance of the application component is created. Apart from the implementation of the scheduling functions, a handler holds performance-optimized structures of the currently processed data.

3.2.1 Data Structure

This section deals with the technical realization of the interlinked data structures in the liveCache. It includes information about the concrete representation of references between persistent objects as well as about how to traverse the network. In **Figure A2** a technical view of the data model is given that shows the concrete implementation of relations from Figure A1.

GUID

The APO distinguishes external and internal references to persistent objects. For the external access from the ABAP layer, a reference is represented by a GUID. Therefore, all persistent objects accessible from the outside are implemented as OmsKeyedObject. Such objects are marked in Figure A2 with the stereotype «keyed». All other objects in Figure A1 are accessed via their containing objects.

The GUID acts as the key here and guarantees the reference is unique in space and time. All links from the outside into the core data model are implemented this way. This solves the problem of possibly reusing the liveCache internal OIDs. Because OIDs are hidden by the API, persistent objects can only be referenced via GUIDs from the ABAP layer.

Anchor Table

Compared with an OID, a GUID does not contain any semantic information about the referenced object, e.g. the type, and therefore is not a suitable standalone object reference. The APO uses GUIDs only at the interface between ABAP and liveCache in anchor tables as a “technical” reference to the persistent objects. Anchor tables store GUIDs together with semantic keys that are used by the ABAP application for accessing objects.

OID

Within the liveCache layer, object references are represented by OIDs. All references that normally would be C++ pointers have to be replaced by OIDs if the referencing and the referenced object are persistent, i.e. links between objects in the APO core data model are OIDs. Figure A2 shows the representation of the relations between core objects at OID level. Navigability of associations means referencing using OIDs.

Lists and Graphs

In addition to simple referencing between two objects, more difficult data structures can be found in the core data model. Examples are single and double linked lists and graphs. The links that form a list or graph are again OIDs.

In Figure A2, the compositions of an activity by its material inputs and outputs are single linked lists. The activity may have links to the first input and output. The material inputs/outputs again are linked in an activity list. Only one link needs to be maintained at a material input/output because it is member of exactly one list of either inputs or outputs.

The lists of allocated and unallocated activities at a resource are examples of doubly linked lists. Activities, together with temporal constraints, form a graph. There are two lists for an activity: one for successor constraints, and another for predecessor constraints. Since a temporal constraint represents the relationship of exactly one successive and one preceding activity, it must be contained in both: the list of successors from the preceding activity and the list of predecessors from the successive activity. Finally, a temporal constraint has links to both activities.

Figure A3 shows an example of an order consisting of seven activities and their temporal constraints. The resulting constraint graph is illustrated by the small picture on the right and can be found again in the thick gray lines lying under the data structure diagram. The arrows between the instances represent links using OIDs. Every order has a reference to its first and last activity. Every activity in turn has a reference to the containing order. The lists of material inputs and outputs belonging to the activities are not shown in Figure A3.

Activity zero and activity six are omitted in the diagram in order to make it easier to read. Therefore, links from their instances are not drawn in Figure A3. Nevertheless, links leading from other instances toward activity zero and activity six are shown by dashed arrows. The link from the order to the first activity would be connected with activity zero. The link to the last activity would be connected with activity six.

Iterators

The concrete implementation of structures like in Figure A3 is encapsulated in order to ease their management and traversal for developers. Each list is accompanied by its own iterator that supports operations needed by clients, e.g. append, insert, remove, search for and access to items. The operations are implemented using generic types that exist for single and doubly linked lists, each in a sorted and unsorted variant. The locking of objects that need to be changed during an insertion or removal is done by the iterators.

Variable Length Objects

In some cases, lists are not suitable for implementing dynamic, array-like structures, e.g. for time lines or bucket vectors. Especially frequent iterations suffer from lost performance caused by dereferencing each single item. Variable length objects are provided by the liveCache. Since such an object can be referenced, locked, and stored only as a whole, the access times for single items are short, at least for more than one access to the same array. Disadvantages are the rough granularity of locks and a large waste of storage for small arrays, due to the fixed size of segments used internally.

Block-Array

The APO uses an alternative approach for storing arrays in the liveCache. **Block-arrays** are part of the APO implementation and not a general feature of the liveCache. Like variable length objects, block-arrays are based on a mechanism that stores segments of the array using ordinary persistent objects. In contrast to variable length objects, the size of the internally used segments can be set by the user, which makes them more suitable for small arrays.

Another difference of block-arrays compared with variable length objects is that single items instead of the whole array are accessed. This may cause a performance loss, if a transaction has to work on many items. The access mechanism is encapsulated in an iterator. An iterator locks only the currently accessed segments, which allows a higher degree of parallel access.

The usage of block-arrays and variable length objects is shown in Figure A2 using the stereotypes «blocked» and «variable».

3.2.2 APO Application Component

This section describes the structure of the APO application component from a technical point of view. The relationships inside the APO application component are depicted in **Figure 10**. The unusual direction of the generalization from top to bottom is due to conformance with the layered structure of the previous diagrams. Figure 10 is a class diagram that shows how the API, handlers and the liveCache are linked together.

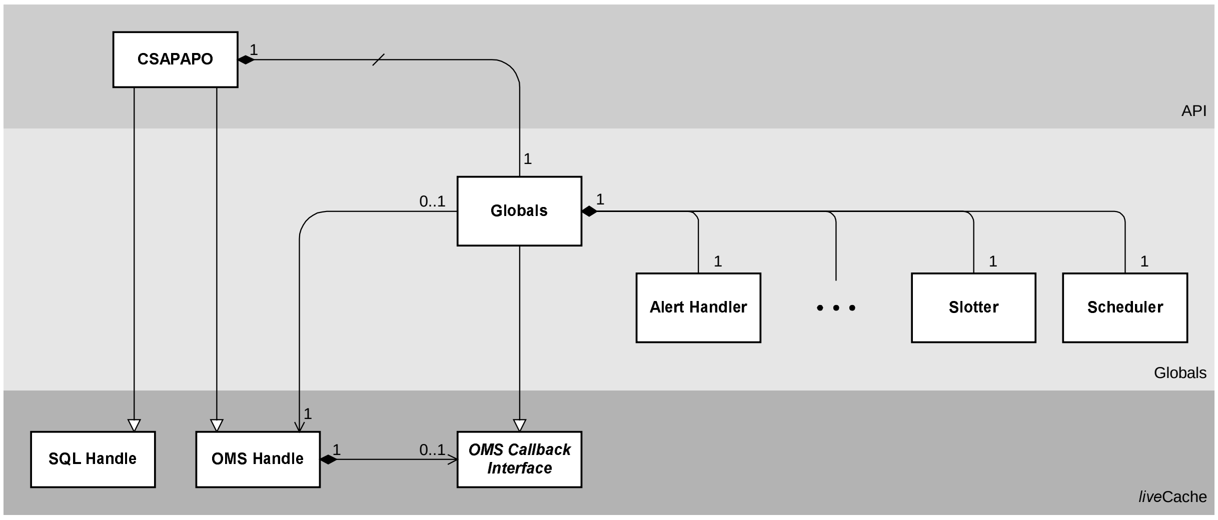


Figure 10 APO Application Component (class diagram)

CSAPAPO

The class CSAPAPO implements the complete API functions of the liveCache layer (for instance the core-, purchase- and transfer-API) except for the ATP. It is derived from the liveCache classes SqlHandle and OmsHandle to get access to persistent and relational data. As explained in Chapter 2, the main application functions are provided by the handlers. Each instance of CSAPAPO contains its own set of handlers.

Globals

In order to simplify the access, all handlers are assembled in one object called Globals. More generally, the Globals object stores any global information, e.g. customizing data. The relation between Globals and CSAPAPO is not established directly. It is derived from the relation between OmsHandle and OmsCallbackInterface. The relation from Globals to OmsHandle enables the handlers to get access to persistent objects.

Callback Interface

Globals is registered for liveCache transaction events on OmsHandle. To enable the reception of those events, it must implement a **callback interface**. For that reason, Globals is derived from OmsCallbackInterface.

ATP

The ATP functions are located in a separate object with relations similar to CSAPAPO in Figure 10. As the ATP check does not use any handler, there was no need to instantiate the Globals except for access to debugging functions and customizing data. Therefore, the ATP does not register an OmsCallbackInterface.

3.2.3 Handler

In this section, the design pattern for the handlers is described. The section ends with a sample sequence of the slotter that illustrates some aspects of the design. As shown in Figure 10, all handlers of an APO application component are assembled in the globals. Each handler has a distinct area of responsibility. It has to process requests concerning different instances of persistent objects. In the APO, a handler is both:

- a proxy for certain types of persistent objects, and
- a function module for manipulating persistent objects

Bridge

The design pattern used for the handlers is known as 'Bridge' or 'Handle/Body'.[17] An implementation of a handler strictly following the bridge pattern is shown in the upper part of **Figure 11**. Every type of handler has exactly one abstraction and several implementors. A client only knows about the abstraction of a handler. The abstraction delegates all requests to the implementors. For that reason, it aggregates all instances of implementors for the particular handler. All (concrete) implementors for one type of handler are derived from the same abstract class.

The design forces a common interface for all concrete implementors, which simplifies their use by the abstraction. The clients are not affected when introducing a new concrete implementor. The changes only concern the abstraction. More concrete, only the code for creating instances has to be adapted at the abstraction.

The main intention for applying the bridge pattern to the design of handlers was to decouple interface and implementation. The resulting consequences are listed below:

- hide the implementation completely from the client (C++)
- break up the application into independent parts to speed up the make process
- allow transparent substitution of implementations
- allow selection of implementations at runtime in order to support the concurrent development process
- cumulate multiple instances of implementations for caching their performance optimized internal structures

[17] Gamma et al. *Design Patterns: elements of reusable object-oriented software*.

The bridge pattern has been extended in some ways to meet the need of the APO as shown in the lower part of Figure 11.

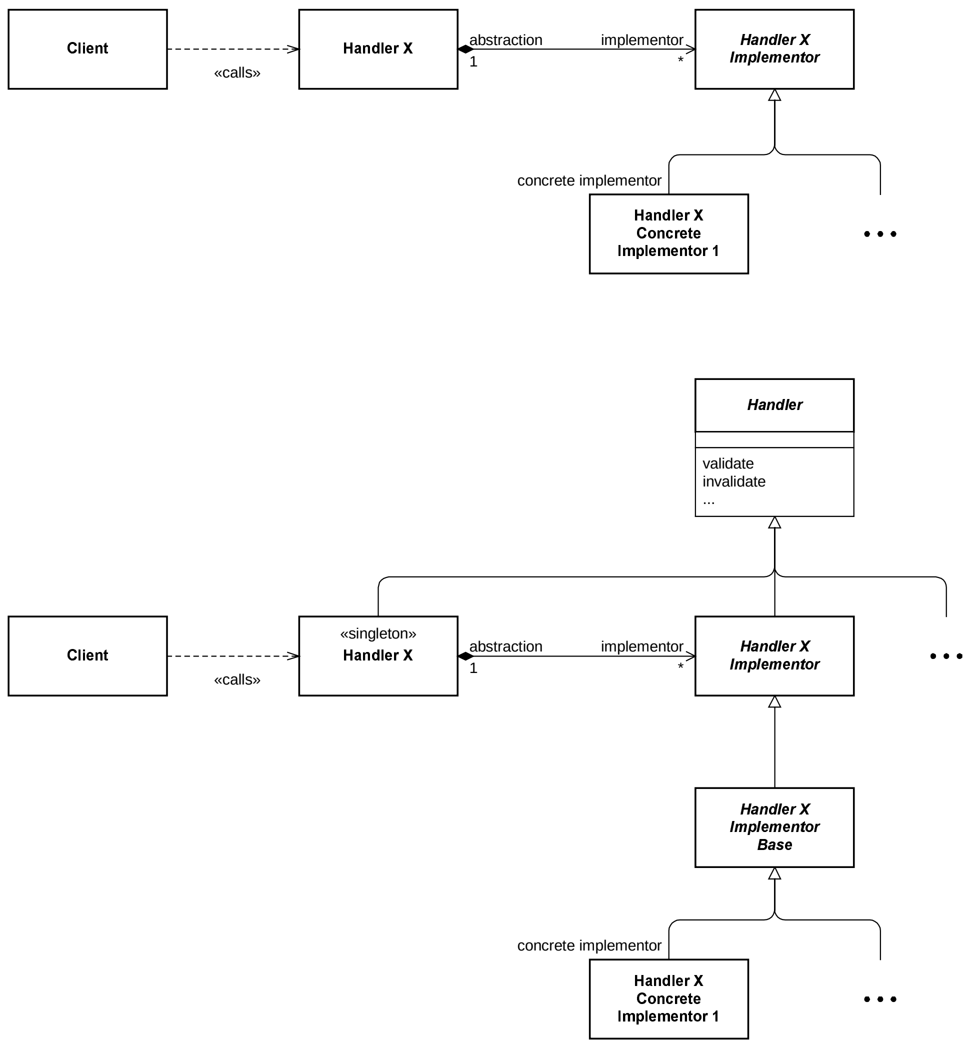


Figure 11 Bridge Pattern and Its Extension

First, all handlers regardless of abstraction or implementor are derived from a single abstract class. This forces a common interface for all handlers, which normally is a contradiction to the intentions above. At the moment, methods for validating and invalidating the internal state of a handler are defined in this class. Because the implementors are not visible for a client, there is no need for having such a common interface.

Some handlers like the slotter introduce an additional level of inheritance between the abstract and the concrete implementors. Common attributes and functions of all implementors are defined in an abstract class.

Singleton

Finally, all handler abstractions in the APO are **singletons**, which means that one instance at most exists per APO application component. Thus, the handler abstractions provide a centralized access point to the core functions. Since an abstraction is a composition of all of its implementors, it has exclusive knowledge of them. It is responsible for management functions like creation, validation and invalidation. Due to the moderate complexity of the creation of concrete implementors, the abstractions come without a dedicated factory instance.

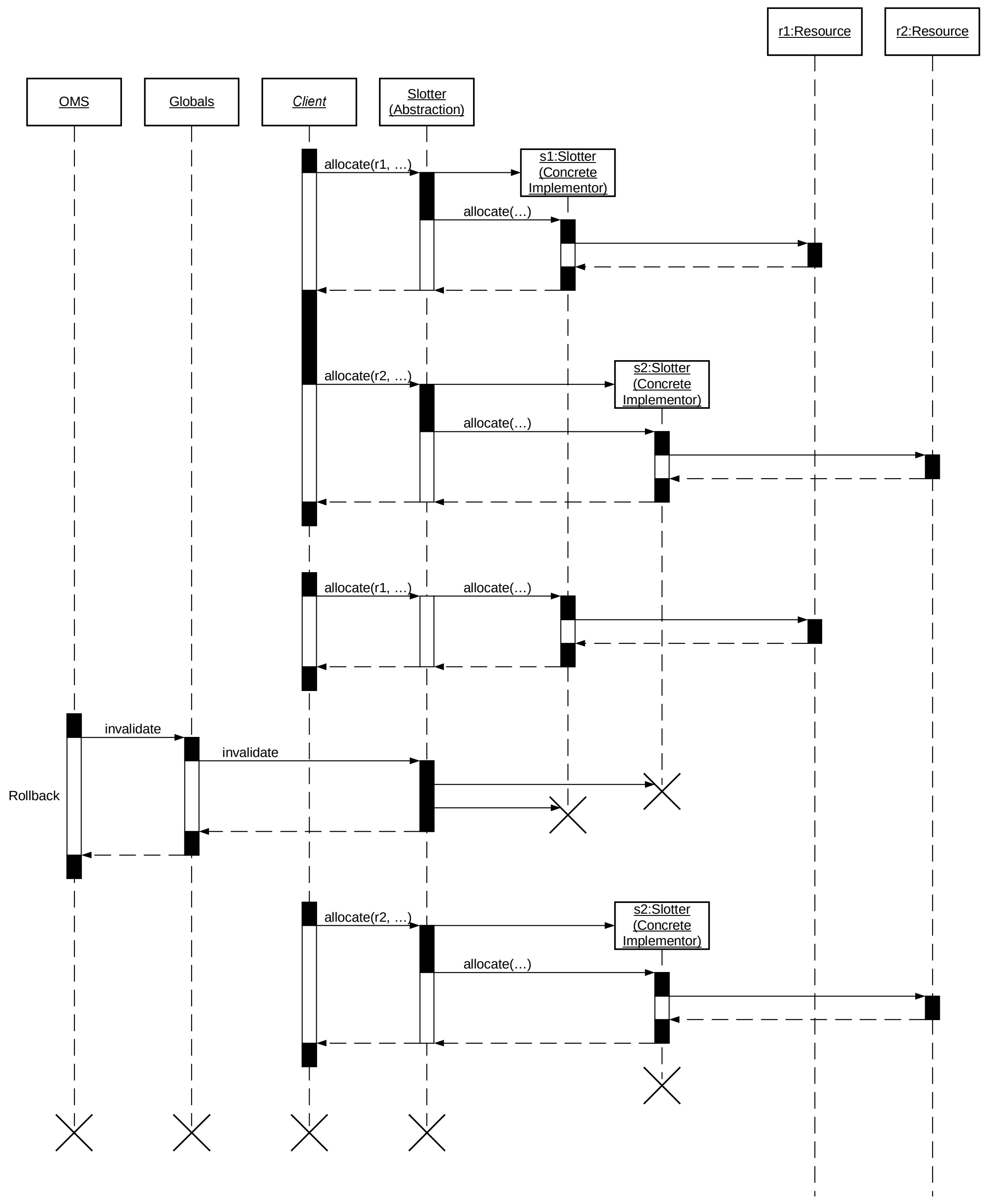


Figure 12 Slotter Abstraction and Implementors (sample sequence)

Sample Scenario

In **Figure 12**, a sample sequence of the collaboration between abstraction and implementors is depicted. A client (the API or another handler) requests the allocation of an order from the slotter abstraction. The request is parameterized with the resource the order should be allocated to.

For each resource accessed, the slotter abstraction maintains a different concrete implementor. If the appropriate implementor already exists, the abstraction passes the request. In the other case, a new concrete implementor is created. During their work, the concrete implementors change attributes at resources. Of course, the slotter makes changes to further objects while allocating an order. These activities are not drawn in Figure 12 in order to keep the diagram simple.

Inside the concrete implementors, performance-optimized representations of the persistent objects are built, which are worth keeping for several requests. Thus, each handler abstraction accumulates several concrete implementors over time. They may become invalid if a new liveCache consistent view, e.g. a new transaction, was started. In this case, the abstraction receives an event to invalidate its current state. Depending on the type of handler, either the implementors are destroyed, or the request is subdelegated.

A more complete information about transaction control in the liveCache is given in the following section.

3.2.4 Transactions

In this section, the usage of liveCache transactions and subtransactions is explained. A liveCache session can be divided into transactions using the SQL commands commit or rollback. A transaction can be split up further with the commands start subtransaction, commit subtransaction, or rollback subtransaction. In the APO, transactions and subtransactions are used according to the process model in **Figure 13**.

For the APO, transaction control remains exclusively in the responsibility of the ABAP application programs. Commit and rollback commands are called from there in order to trigger the operation in the liveCache. The liveCache commit/rollback is independent from the R/3 database. The APO uses the pessimistic transaction mode of the liveCache where locks have to be acquired by an explicit call.

Application Method

According to a convention, every method in the APO application component is embraced by a separate subtransaction. For methods for bulk processing, each task is processed in its own subtransaction. After the conversion of the call parameters, every method starts a new subtransaction. Subsequently, the actual planning and scheduling task is processed. It is finished with a commit subtransaction or rollback subtransaction command depending on the result of the task. For bulk processing, the last two steps are repeated for each task. The method returns after converting the results for the calling ABAP program.

As mentioned in Section 3.2.3, the APO application component, especially the handlers, build their own internal data structures. The data in the handlers is initially based on the consistent view in the private OMS cache but may be modified later. Hence, a handler needs to be informed about commit and rollback commands in order to synchronize its state with OMS cache. The APO application component registers an OmsCallbackInterface to enable the reception of such transaction control events.

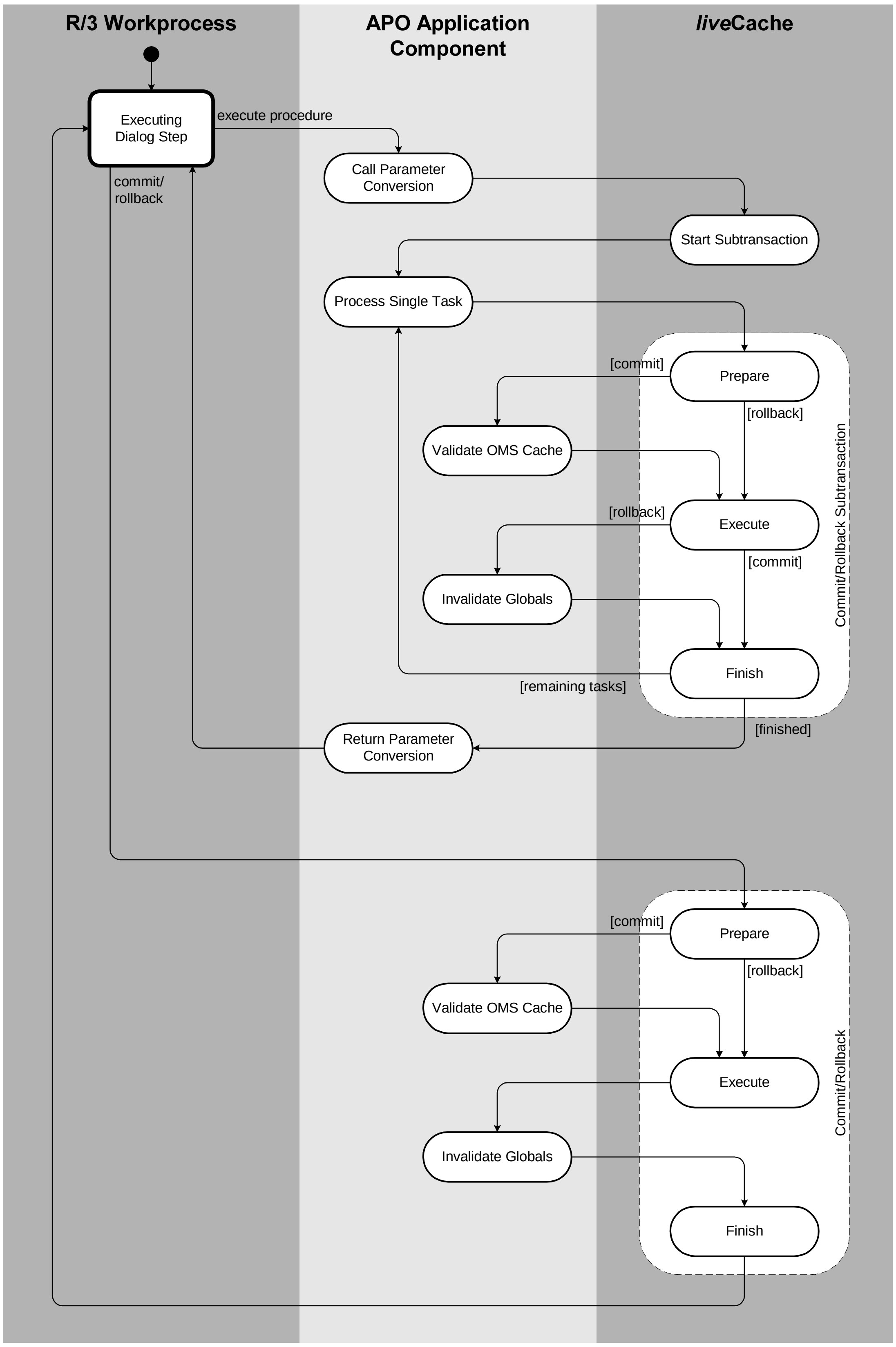


Figure 13 APO liveCache Session

Commit Subtransaction

In case of a commit subtransaction command, the liveCache OMS performs a callback before the actual execution starts. The handlers are responsible for **validating** the state of the OMS cache according to their internal data.

After return of the callback, the OMS makes the changes of the subtransaction durable, at least up to a rollback of the top-level transaction. Since the consistent view is not changed by a commit subtransaction command, the state of the handlers and the OMS cache are valid after its execution.

Rollback Subtransaction

At a rollback subtransaction command, all current changes are discarded, and thus no validation of the OMS cache is needed. After the execution, all changes to the OMS cache are overwritten with the values from the beginning of the subtransaction. A callback is necessary to inform the handlers which in turn **invalidate** the state of their internal data.

The duration of a liveCache transaction is determined by the ABAP application program. More than one successive call of a method of an application component may occur in the same transaction. However, an implicit commit may occur, due to the dispatching of dialog programs between work processes.

Commit/ Rollback

Callbacks comparable to subtransactions are triggered during commit and rollback of top-level transactions. The validation of the OMS cache is also necessary before committing the entire transaction. In any case, the handlers have to be invalidated after the execution of a commit or rollback. The reason is that, in contrast to subtransactions, a new consistent view is started.

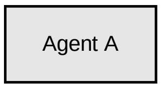
Appendix

A. Modeling Techniques Summary

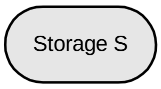
This appendix gives a short overview of the elements in the structure plans used by the Basis Modeling group. For detailed information see the Bluebook **Description Method - Modeling Principles and Structure Plans**.

A.1 Block Diagram

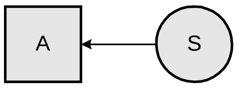
Block diagrams show the structure of a system, i.e. its *components* partitioned in active and passive parts and the *connections* between them. They show the *data flow* in a system.



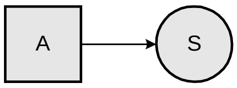
Agent A is an active system component. An agent can be refined with any complex block diagram (i.e. can contain agents and storage places).



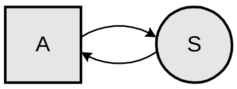
Storage place S is a passive system component. A storage place can be refined with a number of sub-storage places.



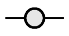
Agent A can read from the storage place S.



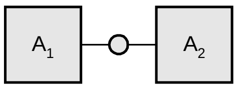
Agent A can write to, but not read from, the storage place S. This can mean a complete or a partial access to the contents of S.



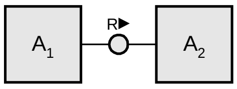
Agent A can write to and read from the storage place S. This can mean a complete or a partial access to the contents of S.



A communication channel is a non-saving, passive component. It can temporarily hold data and events for the duration of a communication.



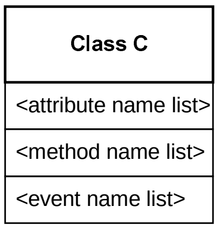
Communication between agent A_1 and agent A_2. The kind and direction of the communication are not specified in more detail.



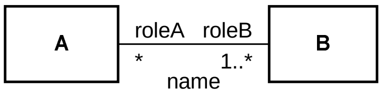
Refinement of the communication channel between two agents. Agent A_1 sends requests (R) to agent A_2 and receives its responses.

A.2 UML Class Diagram

UML class diagrams visualize object classes (often called entity types or object sets) and relationships between them.



Class C. The name is the term of one instance of the class (e.g. the class of all cars is named car). The compartments specifying **attributes** (possibly with type specification) and **methods** (possibly with signature specification) are optional.



Association. Instances of class A are associated with (UML: 'linked to') instances of class B. An instance of class A is associated with at least one instance of class B. The number of instances of class A associated with one instance of class B is not specified. In this example A plays roleA and B plays roleB in the association. Name, roles and cardinalities are optional.

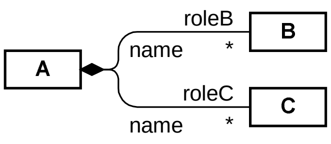
- basic cardinalities:

N	a number n (as variable) or constant (e.g. 5)
*	any number

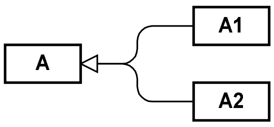
- complex cardinalities:

x..y	Cardinality range, where x and y are basic cardinalities.
a,b,...	Cardinality list, where a,b, etc. are basic cardinalities or cardinality ranges.

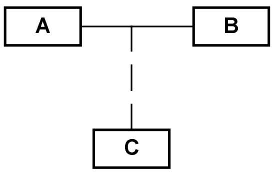
Cardinalities should be expressed as simple as possible, e.g. '*' instead of '0..*'.



Composition. It is a special kind of association. Every instance of classes B and C is associated with exactly one instance of class A. If an instance of class A ceases to exist, the associated instances of classes B and C are destroyed (existence dependency). Name, roles and cardinalities are optional.



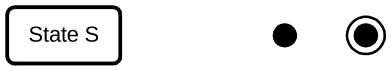
Specialization. The classes A1 and A2 are subclasses of class A.



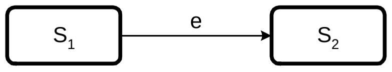
Objectified Association (UML: 'Association Class'). The elements of the association between the instances of classes A and B are viewed as instances of class C. Thus they can have attributes and methods, can be associated with instances of other classes, and be specialized.

A.3 UML Statechart and Activity Diagrams

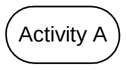
Statechart diagrams show system/object states and events leading to state transitions. An activity diagram is a variation of a statechart diagram, representing all possible sequences of operations in a system or system group. Statechart diagrams consist of states and transitions. The states are represented by state symbols and the transitions are represented by arrows connecting the state symbols. States may also contain subdiagrams by physical containment and tiling. An activity diagram is a special case of a state diagram in which all (or at least most) of the states are activities (action states). A transition leaving an activity is implicitly triggered by completion of the source activity. A statechart diagram shows the state transitions of only one system/object whereas activity diagrams may represent the interaction of several systems/objects.



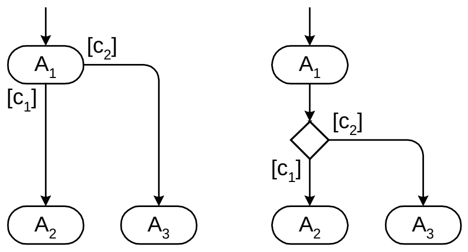
Simple state S, *initial* and *final* (pseudo) state. Inside the rectangle, the name of the state may be shown.



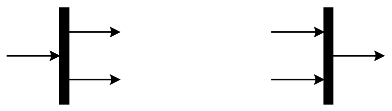
Transition between states S_1 and S_2 triggered by the event e. The text near the arrow denotes the event that triggers the transition.



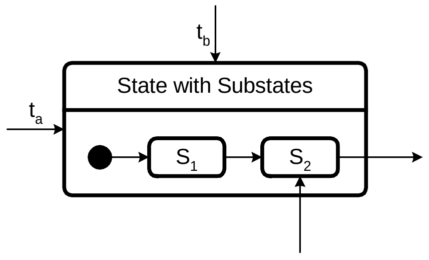
Activity A (action state). The text in the activity describes the actions executed at an arriving transition.



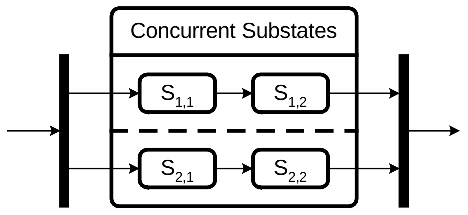
Decision. Multiple output transitions of activity A_1 are labeled with different guard conditions c_1 and c_2. Depending on which condition is fulfilled after execution of A_1, the transition to activity A_2 or activity A_3 is triggered. The figure on the right shows an alternative representation with a *branch* (pseudo) state depicted as a diamond.



Fork/Join. The *fork/join* (pseudo) state is shown as a heavy bar and represents either a splitting of control into concurrent threads (left picture) or a synchronization of concurrent threads (right picture).

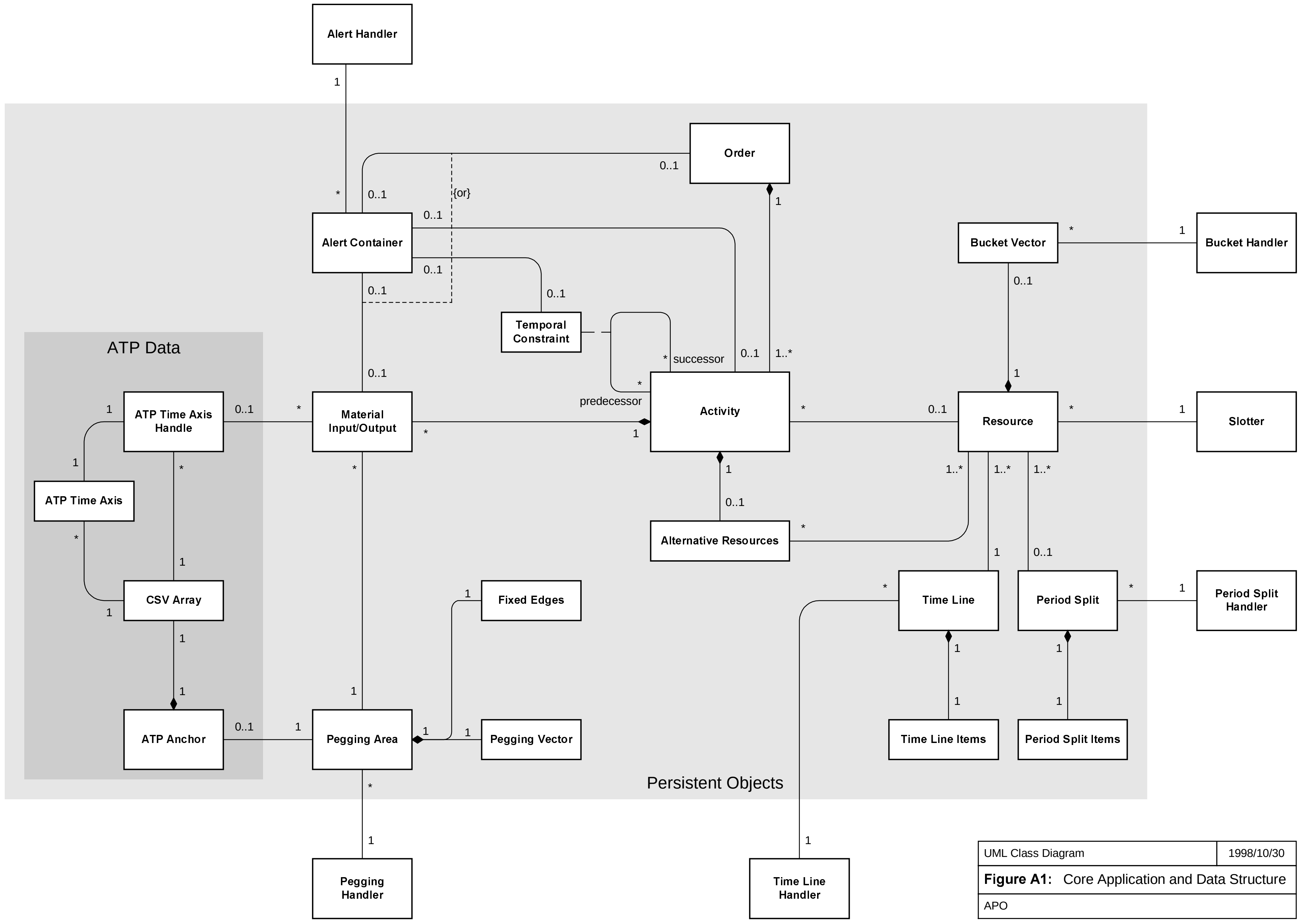


Composite state. A state can be decomposed into mutually exclusive disjoint substates. The transition from the *initial* state inside the composite state to the state S_1 is triggered by any transition to the enclosing composite state, e.g. transition t_a and t_b. Transitions may be drawn directly to or from states enclosed in a composite state.



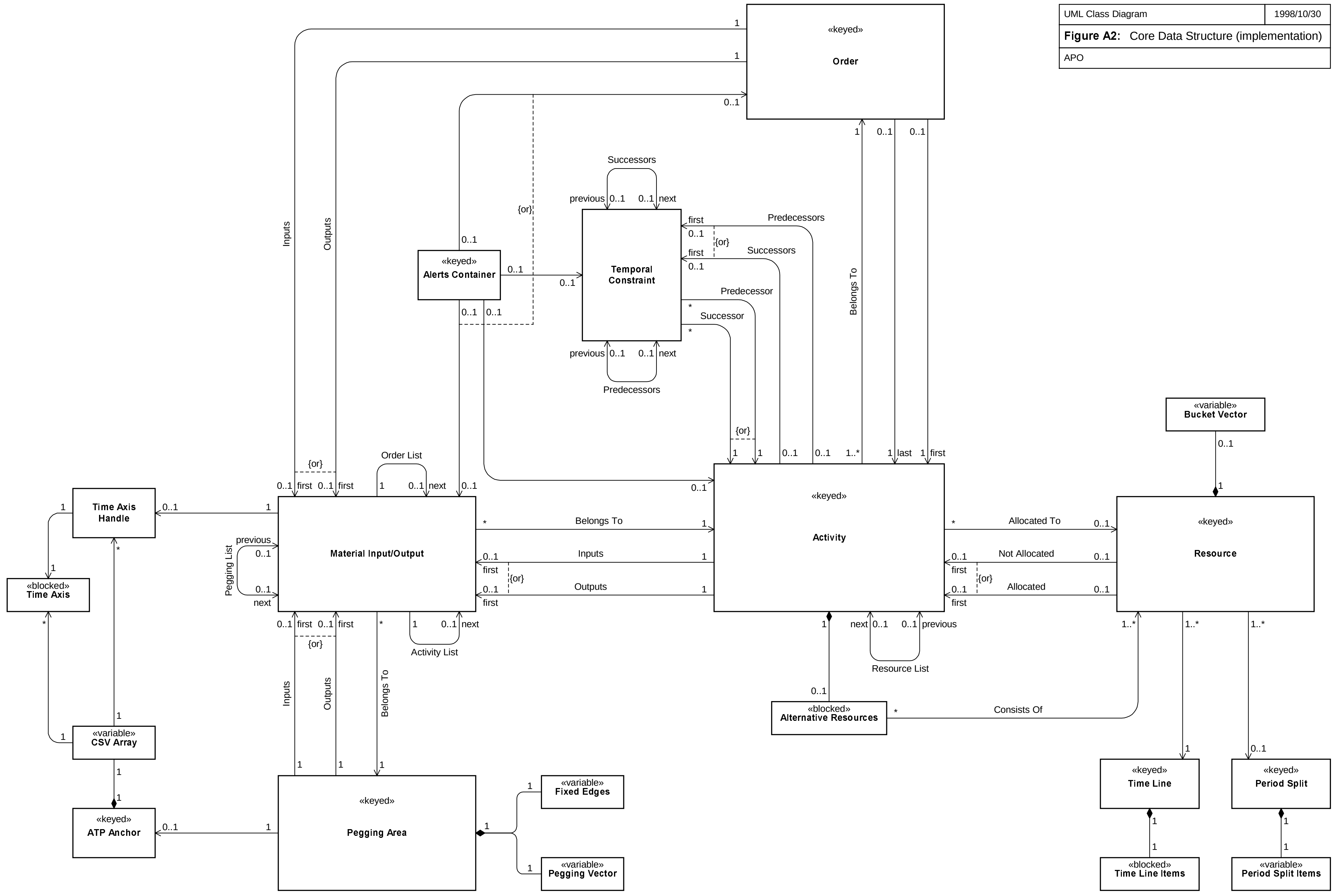
Composite state with concurrent substates. The lower compartment containing the substates is divided into two subregions using a dashed line. Each of those regions is a concurrent substate. The states inside each subregion are again mutually exclusive.

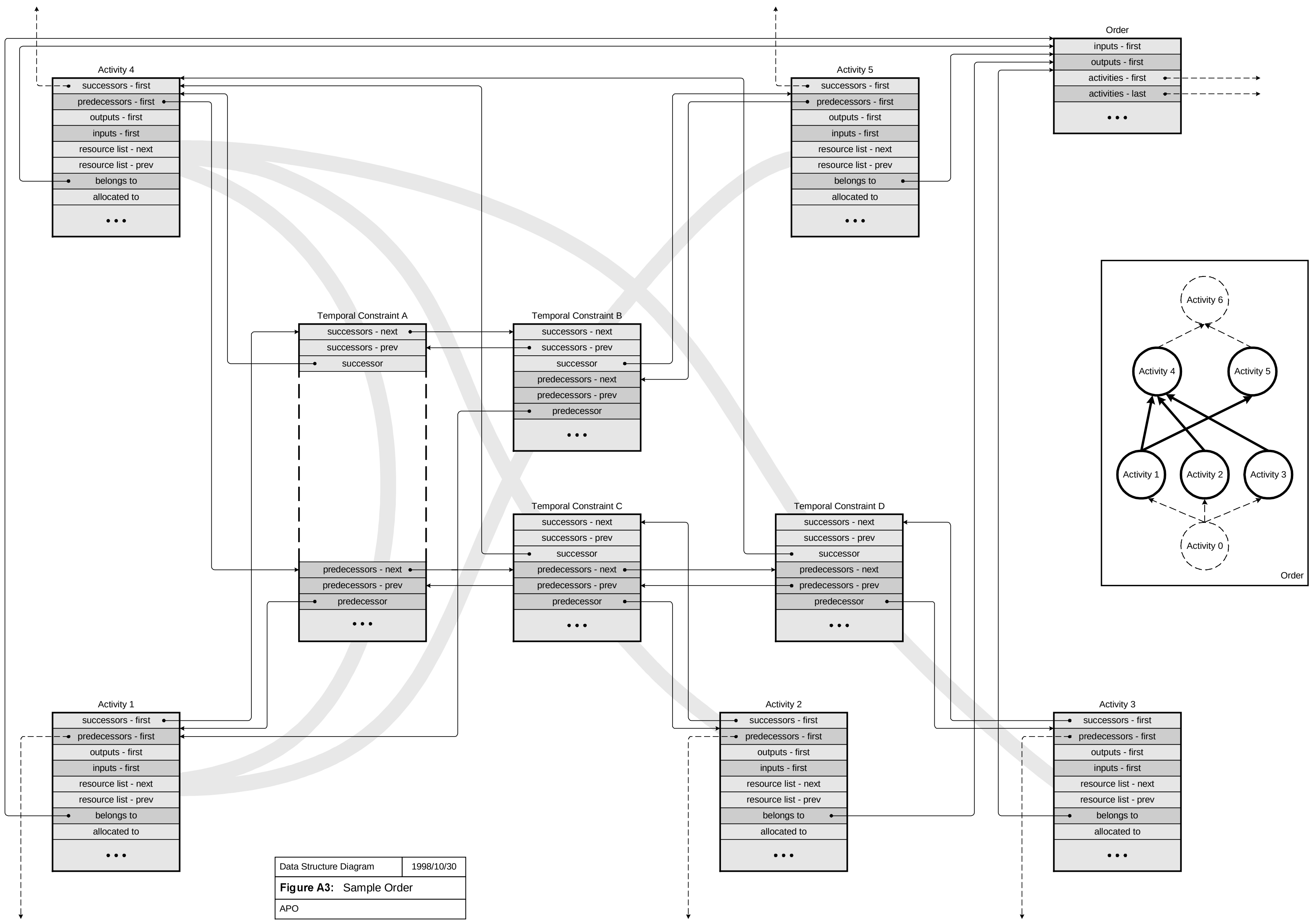
Index



UML Class Diagram	1998/10/30
Figure A1: Core Application and Data Structure	
APO	

UML Class Diagram	1998/10/30
Figure A2: Core Data Structure (implementation)	
APO	





Data Structure Diagram	1998/10/30
Figure A3: Sample Order	
APO	