# APEX Dynamic Controller (ADC) – Technical Documentation

The APEX Dynamic Controller (ADC) is an APEX dynamic action plugin for creating dynamic APEX application pages without the overhead of many dynamic actions otherwise required to manage the state of form elements.

The plugin controls the application page by setting up use cases in a separate APEX application. This application manages the use cases for all APEX applications of a workspace that are to be extended with ADC. A use case follows a specialized version of the IFTTT pattern (IF THIS THEN THAT) by focusing on the application user:

IF the user does THIS, THEN do THAT.

Any number of actions may be defined for one use case which may result in actions both within database and on application side. An action may, for example, change application's session state (or application state for short) within database as well as control visibility of application's elements on APEX application's side. For this purpose, the plugin provides a selection of predefined action types, which can be extended by the developer according to her own needs.

Note:

The session state varies from the application state in that the term application state refers to the actual state of the input items on the application page. The session state may be out of sync with the application state after the user changed data or because input item values are not saved in session state at all. So to be clear about this difference, application state always means the actually valid item values.

If a use case changes the application state, this can have the effect of triggering another use case. In this way, the use cases can be formulated in a precise and compact manner without having to traverse between client and server multiple times. All use cases are processed recursively in one operation and a combined response with instructions for the application side is compiled.

The plugin has the following functionality:

* Management of the visibility status of form elements (active, deactivated, visible, hidden)
* Validation of input, output of corresponding error messages
* Application state management, determination of element values based on the application user's selection
* Recursive rule triggering
* Easy administration: the plugin does not require any administration on the application side, all necessary event handlers, selection of element values to be transferred and all other configuration is automatically derived from the entered rules

All functions of the plugin are offered dynamically via AJAX calls, there is no reloading of the application page.

**Plugin concept**

The plugin implements use cases using a decision table that evaluates the current application state against the defined use cases. The application state contains the current user input, along with meta information about the triggered event (which element triggered the event, which event was triggered, etc.). When the plugin is called, the application state is sent to the controller.

The use cases are implemented as a rule of the decision table where the IF THIS part is expressed by a WHERE clause (technical condition). The technical condition has access to the application state by providing it as columns of the decision table. Thus, if an input field named P10\_ENAME exists on the application page, a column P10\_ENAME with the current session value is offered to the decision table. Necessary type conversions to number or date values have already been done. This way a date column can be compared directly against e.g. SYSDATE.

Example:

The use case "The user creates a company address" is expressed by the technical condition P10\_ADDRESS\_TYPE = 'J'.

The plugin determines the first matching use case from the given application state and executes the actions stored in that use case. The order of the individual use cases can be controlled to execute more specific use cases before more general use cases for the same application state.

A use case consists, besides the technical condition, of a collection of actions to be executed. The plugin provides a predefined set of action types that are executed either within the database as PL/SQL actions, or in the browser, as JavaScript actions. Custom action types can be created by metadata. Here, the identifier of the action type, a possible PL/SQL activity and/or a JavaScript method must be defined. Any parameters of the action type can be freely defined (up to three parameters per action type) and referenced by replacement strings in the PL/SQL and JavaScript methods.

The plugin transmits only the relevant application state each time it is called. The page elements that are addressed in at least one technical condition of the use case are considered "relevant to the application state". So, if a technical condition is defined as P1\_VALID\_FROM < P1\_VALID\_TIL, plugin will recognize both elements P1\_VALID\_FROM and P1\_VALID\_TIL as relevant. Extracted elements must be present on APEX application page, or available as application elements. If an element is used that is not present, the technical condition does not validate.

The fact that the relevant elements are extracted has the consequence that...

* these elements are provided by the plugin with an event handler for the CHANGE event (or CLICK event if it is a button). This triggers the processing of the changed inputs of these elements or the click on the button by the plugin.
* the current element values of these elements are copied to the application state when the plug-in is called. This makes the current element values of the SQL decision table available for filtering by the technical conditions.

Should plugin change application state, these will be passed to application in plugin's response and will be inserted by plugin into corresponding form elements. If relevant application elements are affected by such a change, associated use cases will be processed recursively.

## Differentiation from Dynamic Actions (DA)

At first glance, the similarities to dynamic actions are quite large, but there are fundamental differences:

* ADC works server-side, DA client-side. Since most use cases require data from the database (for example, for validation or decision making), ADC is more favorable in these deployment scenarios
* ADC integrates binding to page events, in DA a matching event must be bound. This reduces the risk of hard to find errors due to incorrect binding to events
* ADC focuses on business logic for managing page state, DA are a general purpose weapon for simplifying client-side code. By focusing on this purpose, ADC is significantly easier to use than DA
* ADC processes the rules in a set-oriented manner and supports recursive rule execution, DA check "one by one" and are in principle not able to execute recursion in the database
* ADC usually requires no JavaScript programming and no client-side code by the user, DA must at least be set up and bound client-side.

From this comparison no criticism of Dynamic Actions can be derived, they are simply intended for a more generic purpose and already from there in the concrete use in a given scenario not always easier or better than a specialized solution.

Simple Dynamic Actions can be realized faster and easier with ADC, because the binding of the event and the problems associated with it are not necessary, furthermore the creation of a rule is faster and easier than the creation of a Dynamic Action.

## Use of the plugin

This section describes the basic operation and use of the ADC plugin.

### ADC Plugin

In the APEX application to be controlled by ADC, installation of a Dynamic Action plug-in is required. It contains two application-level parameters that must be set before use.

#### Parameter »ADC APEX Functions«

The plugin supports usage with several APEX versions and own or modified themes. The required adaptation of ADC functions to APEX version or used theme is done by JavaScript file for corresponding combination of theme and version. ADC provides an adapter file for use with Theme 42 in several versions. Versions are realized as namespace objects within that adapter file and can be selected by their namespace using that parameter. For example, if you are using APEX in version 20.2, you can select the entry de.condes.plugin.adc.apex\_42\_5\_1 here. As an advanced use of the plugin, you can write your own adapter files. This is necessary if you …

* need to provide appropriate functionality in JavaScript for custom action types
* use a heavily customized theme that does not work correctly with the default functionality of the plugin.

As a rule, you inherit a namespace object that is as suitable as possible and extend it with the changed or new functionality.

#### Parameter » Output comments «

This parameter controls whether you want to output advanced comments on the browser console when working with SCT. This functionality is especially useful on development machines to monitor correct operation. On production servers, this setting can be disabled to save bandwidth.

#### Integration of the plugin

The plugin only needs to be called on the application page when loading the page, no parameterization is required. As an alternative to including the plugin on each page to be controlled, it can also be inserted on page 0 and thus included for each application page. A server-side condition can prevent the execution of the plugin for a list of application pages.

### Use case capture

Use cases are captured in the separately supplied APEX application. If the plugin is installed on an application page and this application page is called for the first time, an entry for this application page is added in the ADC data model.

Even without recording use cases for an application page, it already behaves differently than a static APEX application page. In particular, all mandatory fields (marked by a label template containing the label REQUIRED) are already checked dynamically. If an input field is filled and then emptied, a generic error message is issued when the input field is exited. In addition, input fields that contain a number or date are dynamically validated to determine whether they contain a corresponding data type. NULL values are tolerated. The conversion of number and date fields is done by the stored format masks, if available, in case of a date field alternatively by the date format of the application..

Note:

When a dynamic application page is created, it should be fully responsible for validating the application page. In particular, make sure that the option that requires an input field is deselected to avoid unpredictable behavior. Also, these pages should no longer contain Dynamic Actions that change the state of the application pages. In this context, also dispense with cascading selection lists, master-detail reports with declarative updating, and similar settings. Since these settings are executed via event handlers on the application page, mutual influences cannot be ruled out and also make it difficult to understand the processes on the application page..

Now use cases can be captured for this page. Predefined is a use case "the page opens" with the technical condition INITIALIZING = C\_TRUE.

Note:

The use case identifiers should be named in the context "When the user ..." to implement the IFTTT concept.

As described earlier, the application status includes meta information about the event that occurred. When the page is loaded, no user event (such as changing an input field or clicking a button) is present. This is indicated by the INITIALIZING column, which in this case contains the value C\_TRUE. C\_TRUE and C\_FALSE are further columns with truth values that you can use for checks. Which values these columns represent depends on the installation of the ADC (see Installation). Similarly, other columns are provided, for example, one column each for the buttons that contain the value C\_TRUE when clicked, or a column FIRING\_ITEM that contains the name of the triggering page element. In the case of initializing the page, FIRING\_ITEM contains the DOCUMENT value. A complete list of columns can be found in the appendix.

A use case has a designation and a technical condition, which are closely intertwined. The technical condition defines how a use case can be recognized from the data of the application. This can be a simple comparison of an input field with a constant value, but also the result of a function call in PL/SQL, which was calculated for the current application state..

Since the syntax of the technical condition corresponds to a WHERE clause, any complex Boolean expressions are possible. However, complex conditions should not be expressed here, but encapsulated in a function. The usual concerns about calling PL/SQL functions in the WHERE clause of an SQL statement do not apply here, because it is ensured that the SQL query always contains only one row, which is checked by the rules. This is because the query always refers to the application state, which contains only one row. Environment changes are therefore only to be expected in small numbers.

Any number of actions can be entered for a use case, which are executed in the order in which they are sorted. The action types support the entry with precisely fitting parameters as well as with help texts that are dynamically displayed for each action type and explain the various options.

Once use cases and actions have been captured, they will take effect immediately. When capturing a new use case, it may be necessary to reload APEX application page to make sure all event handlers are bound correctly. Changes in actions do not require reloading the page, but are taken into account immediately.

### Action types

Use cases reference parameterized action types to perform the desired activities within the database and/or on the application side. These action types are defined in a normalized data model and include:

* A technical identifier, a plain text name, and a representation text for the THEN branch of the IFTTT pattern.
* A help text describing the usage
* One PL/SQL and JavaScript code block each with the replacement anchors ITEM, PARAM\_1 … PARAM\_3
* A reference to a subset of the page items relevant for the current action type (e.g. All input fields of the current page), which is offered as a pop-up list during parameterization
* None or up to three parameters, each with the following definitions
  + Type of parameter (restriction of possible values, partly via pop-up lists, e.g. all regions of the current page).
  + Help text
  + Default value
  + Flag to indicate whether the parameter is a mandatory value.

The correspondingly parameterized actions, based on the action types, are assigned to a use case and executed accordingly. The developer can create his own action types and thus extend ADC.

### Logical building blocks and data model

ADC is implemented as a stand-alone APEX application and therefore requires its own data model as well as a plugin to be installed separately for the APEX applications to be controlled by ADC.

#### Naming conventions used

The data model uses a naming convention which on the one hand keeps the related tables together, on the other hand is optimally prepared for APEX applications and furthermore facilitates comprehension.

All table names start with the abbreviation ADC. Each table defines a three-letter abbreviation that precedes each column name. The abbreviations are specified when discussing the individual tables.

Each table includes a <table abbreviation>\_ID column as a primary key, if meaningful. Foreign key relationships are named in the form <table abbreviation>\_<referenced table abbreviation>\_ID. This way, the function can be read from the column name, moreover all column names are unique.

Constraints are named according to two conventions:

* Primary key constraints have the name <table name>\_PK
* - All other constraints have the name <column name>\_<constraint abbreviation>.

Constraint abbreviations can be for example CHK for check constraints or FK for foreign key constraints.

#### Rule group, table ADC\_RULE\_GROUPS (CGR)

If on an APEX application page the control of the status of form elements, regions and buttons is to be taken over by the plugin ADC, this has to be stored in the meta data of ADC.

Note:

A dynamic application page is internally referred to as a rule group, and the use cases as rules. This is due to the technical implementation as a decision table.

One regular group includes

* a technical ID
* a reference to the APEX application and page, each via its ID
* a column to hold the Decision Table logic
* a column to hold any initialization logic
* Flags to control behavior, such as whether a rule group should be executed or not and whether recursive execution is allowed.

The task of the rule group is exclusively the grouping of rules. It represents an area of responsibility within which rules and their dependencies are analyzed. Rule groups are stored in the ADC\_RULE\_GROUPS table.

#### Rule, table ADC\_RULES (CRU)

The rule group is assigned rules in the following, which in turn consist of a technical ID, a descriptive name, a condition and a sort criterion. The rules reference the rule groups they belong to by their technical ID and also serve as a reference for any number of actions to be executed when the rule's condition evaluates to TRUE and the plugin selects this rule for execution.

In addition, ADC maintains for each rule a list of the names of the form elements or buttons that are referenced in the condition. The list of these referenced elements in their entirety represents the list of "relevant" elements in that rule group. Rules are stored in the ADC\_RULES table.

#### Actions, table ADC\_RULE\_ACTION (CRA)

Each rule is assigned actions to be executed when the rule evaluates to TRUE and the plugin selects this rule for execution. The actions reference an action type stored in table ADC\_ACTION\_TYPE, as well as the rule and rule group to which they belong. In addition, they reference a page element to which the action refers. For example, if the value of the form element P1\_VALID\_TIL is to be set to the value NULL, this action references the element P1\_VALID\_TIL and calls the action type SET\_NULL, which in turn defines how an element value is to be set to the value NULL. The actions also have a sort criterion that controls the order in which actions are to be executed.

Actions contain three optional attribute values in the PARAM\_1 … PARAM\_3 columns. These attributes can be referenced by the action types (see there) and are used to pass parameters to the PL/SQL or JavaScript code.

Actions are stored in the ADC\_RULE\_ACTIONS table. The parameterization of an action type can include up to three parameters, which are stored as a normalized relationship in the ADC\_ACTION\_PARAMETERS table. Thus, in principle, it is also possible to define more than three parameters for an action type, although this is currently not supported by the user interface.

#### Action types, table ADC\_RULE\_ACTION\_TYPES (CAT)

An action has an action type. This defines how an action is to be implemented. An action type includes a technical, alphanumeric key, a reference to a translatable element for display on the interface, an implementation statement for the PL/SQL language, and an implementation statement for the JavaScript language. The implementation statements are optional and may not be set at all, only in PL/SQL, only in JavaScript, or in both languages simultaneously. Statements must always be written as a snippet from an anonymous PL/SQL block or a JavaScript block, so in particular they need a semicolon at their end.

The plugin collects all PL/SQL statements and executes them within the database, then the changed element values are loaded from the application state and sent with the JavaScript statement to the APEX application where they are executed. This results in a fixed execution order:

* all PL/SQL statements in the order defined by the SORT\_SEQ column of the ADC\_RULE\_ACTION table.
* all JavaScript statements in the order defined by column SORT\_SEQ of the ADC\_RULE\_ACTION table.

Action types are allowed to reference replacement strings that will be filled with transaction data when used later on:

* #ITEM#  
  This replacement anchor is filled at runtime with the name of the referenced object from column CRA\_CPI\_ID of table ADC\_RULE\_ACTION
* #PARAM\_1# … #PARAM\_3#  
  These replacement anchors are filled at runtime with the CRA\_PARAM\_<n> columns of the ADC\_RULE\_ACTIONS table. They are used to pass parameters to PL/SQL or JavaScript functions, e.g. for message texts etc.
* #SELECTOR#  
  This replacement is used only for JavaScript action types. If used, it analyzes whether a page item is referenced or a jQuery expression is passed in.

Action types are stored in the ADC\_ACTION\_TYPES table. The structure of an action type is normalized and includes additional tables:

* Table ADC\_ACTION\_ITEM\_FOCUS  
  This table defines subsets of addressable element types on an APEX page or an APEX application. The subsets are defined as a list of element types according to the following specification:
  + DOCUMENT (no triggering element or use of a jQuery selector).
  + APP\_ITEM (application element of the current application)
  + BUTTON (button of the current application page with static ID)
  + ITEM (input field of the current application page)
  + REGION (regions of the current application page with static ID)
* Table ADC\_ACTION\_PARAM\_TYPES  
  Defines parameter types that are assigned to a concrete parameter of an action type. Based on the parameter type, the interface controls which input field is displayed. For example, a list of all available PIT messages could be displayed for the parameter type PIT\_MESSAGE, while an input field for text is stored for the parameter type JAVA\_SCRIPT. Based on the parameter type, ADC can perform plausibility checks of the inputs.
* Table ADC\_ACTION\_TYPE\_GROUPS  
  Action types can be grouped by entries in this table. This grouping refers to the selection list of action types and combines different action types under one group identifier. For example, all action types that refer to input fields could be grouped together in a "Actions for input fields" group. The grouping has no effect on the function of the action types.

#### Page elements, table ADC\_PAGE\_ITEMS (CPI)

Rule conditions and actions reference page elements. The referenced page elements are read and analyzed from the APEX Data Dictionary for the APEX application referenced in the rule group. The prepared descriptions of these page elements are then stored in table ADC\_PAGE\_ITEMS. Each page element is described …

* by a reference to the rule group from which the page element is referenced
* by the ID of the element (for form elements this is the name of the element, for buttons and regions only the static ID assigned on the APEX application is referenced)
* a reference to the type of element (page element, application element, button or region) stored in table ADC\_PAGE\_ITEM\_TYPES
* as well as information about the conversion of the element into the data types TEXT, DATE or NUMBER

The table includes all elements of the application page. Application elements which have been referenced in a rule condition are marked by column value CPI\_REQUIRED = C\_TRUE, because these elements are needed for rule evaluation. Actions can change all elements of APEX application, but these changes will cause plugin's reaction only if changed elements were used in a rule condition.

The conversion of an element to the data types DATE or NUMBER is only recognized if the element has a format mask on the application page or contains this information via the use in a form region via the metadata. Therefore, the following rules must be observed if the plugin is to be able to cooperate with an application page without any problems:

* Elements to be converted to NUMBER or DATE must have a format mask in APEX application or be used via a form region.
* If page elements in APEX do not have possibility to define format masks, these elements have to be explicitly transferred to target data types in rule conditions (e.g. SELECT lists or application elements)
* Buttons or regions which are to be manipulated by actions require mandatory static ID on APEX side, otherwise they won't be recognized by plugin. Buttons can furthermore be used in rule conditions because they are assigned a column with the identifier of the static ID, which contain C\_TRUE if clicked, regions (by their nature) do not
* Starting with APEX 20.1, all page items which are referenced by technical conditions must be set to be stored on disk (as opposed to kept in memory only)

Page elements are stored in the ADC\_PAGE\_ITEMS table.

#### Page element type, table ADC\_PAGE\_ITEM\_TYPES (CIT)

Page elements can be of type APP\_ITEM, ITEM, BUTTON or REGION. These page element types are stored in a separate table, ADC\_PAGE\_ITEM\_TYPES. In addition to technical, alphanumeric ID and reference to translatable object of types, it will be stored for page element types whether these types have session state in APEX and whether and if yes, which event these element types shall bind. For the given element types it is defined that only the types ITEM and APP\_ITEM have a session state and the element types APP\_ITEM, ITEM and BUTTON bind the events CHANGE and CLICK respectively. Regions can therefore neither be used in rule conditions nor can they have a session state. However, regions can be manipulated (e.g. visible or invisible, collapsed or expanded, or updated).

Events can also be stored as special page element types. This makes it possible to react to special events of the application page, such as the event apexafterclosedialog. The focus of this functionality is not to imitate dynamic actions, but to be able to react to events of the application page that change the status of the page. The list of events should be kept as short as possible. If an event is stored in this way, the decision table provides a corresponding column containing the triggering element if the event was triggered and NULL otherwise.

### Processing a rule

The basis of the plugin is the transfer of decision logic into an SQL query. This will be illustrated with an example.

#### Processing example

An APEX application page contains two elements P1\_VALID\_FROM and P1\_VALID\_TIL. It is to be specified that if...

* P1\_VALID\_FROM contains the value NULL, field P1\_VALID\_TIL is to be deactivated
* P1\_VALID\_FROM is greater than or equal to P1\_VALID\_TIL, an error message is to be output for field P1\_VALID\_TIL
* P1\_VALID\_FROM contains a value, field P1\_VALID\_TIL is to be activated

Finally, three rules have been created:

* Condition with ID 5: P1\_VALID\_FROM is NULL  
  Action: P1\_VALID\_TIL: Disable
* Condition with ID 6: P1\_VALID\_FROM >= P1\_VALID\_TIL  
  Action: P1\_VALID\_TIL: Show error message
* - Condition with ID 7: P1\_VALID\_FROM is not NULL  
  Action: P1\_VALID\_TIL: Activate

The order of the rules is important here: The rule with ID 7 must be executed after rule 6, otherwise rule 6 would no longer be considered. The conditions must therefore be created from the specific to the general. Typically, initialization rules are formulated first, followed by error status and finally the behavior in the normal case.

From the analysis of the individual conditions, it can be seen that the elements P1\_VALID\_FROM and P1\_VALID\_TIL have been referenced. Therefore, to analyze the rules using SQL, it is important to know the current value of these two elements. The only source for SQL statements to query the value of application elements is the application state. However, since this is not necessarily synchronous with the application page, the first step is to ensure that the current element values of the application page are copied to the application state.

In order to filter the relevant elements from a rule, a regular expression is used that searches for all the elements in question (all the elements available on the APEX page, the application elements and buttons made available) in the rule conditions.

Subsequently, the value of the application elements can be queried with the following SQL query:

select to\_date(v('P1\_VALID\_FROM'), 'dd.mm.yyyy') valid\_from,  
 to\_date(v('P1\_VALID\_TIL'), 'dd.mm.yyyy') valid\_til  
 from dual;

The plugin creates this query automatically, based on the metadata from the tables ADC\_PAGE\_ITEMS and ADC\_RULE. In addition, the query records other information:

* The triggering element of the request is made accessible by the FIRING\_ITEM column and returns the ID of the triggering element. If the plugin is initialized, this column contains the DOCUMENT value, otherwise the name of the triggering element or the static ID of the triggering button, both in capitals.
* If the triggering element is DOCUMENT, i.e. the plugin is initialized, the column INITIALIZING contains the value C\_TRUE, otherwise C\_FALSE.
* The elements addressed in a rule is available via the FIRING\_ITEMS column in the rule view and is used to evaluate only those rules that have a dependency on the triggering element. For this purpose, the triggering element (FIRING\_ITEM column) is filtered against the FIRING\_ITEMS column using INSTR(). In addition to the addressed elements, the FIRING\_ITEMS column also contains the DOCUMENT entry, so that all rules are evaluated when the page is initialized, since in this case FIRING\_ITEM contains the DOCUMENT value.
* Buttons are also referenced via a column corresponding to their static ID. They get the value C\_TRUE if they were triggering element, otherwise C\_FALSE.

The additional columns are used to simplify the formulation of rules. For example, a rule that compares the page elements P1\_VALID\_FROM and P1\_VALID\_TIL can be delimited by an additional reference to the FIRING\_ITEM column so that it is only applied when the element P1\_VALID\_TIL changes:

P1\_VALID\_FROM >= P1\_VALID\_TIL and FIRING\_ITEM = 'P1\_VALID\_TIL'.

The INITIALIZING column can be used to create rules that are executed only when the application page is initialized: INITIALIZING = C\_TRUE.

Buttons, as mentioned, are also made accessible by columns. For example, if a button exists with the static ID B1\_SAVE, it would contain a column B1\_SAVE that would contain the value C\_TRUE whenever B1\_SAVE was the triggering element, and C\_FALSE otherwise. This can be used to create a simple rule that is applied only when button B1\_SAVE was triggering element: B1\_SAVE = C\_TRUE. Alternatively, the same condition could be formulated using the FIRING\_ITEM column: FIRING\_ITEM = 'B1\_SAVE'.

Columns C\_TRUE and C\_FALSE contain Boolean values as defined in ADC\_UTIL.C\_TRUE/FALSE. What values are returned depends on the FLAG\_TYPE initialization you set when installing ADC.

The application state data is available to the rule view in an inner view named SESSION\_STATE, which is subsequently filtered by the conditions. The rule conditions are structured as follows: ((SRU\_ID = <rule ID>) and (<rule condition>)). All rules are combined by an OR connection in the WHERE clause. This results in the following WHERE condition for our example rule group:

with SESSION\_STATE as (...)  
select ...  
 from SESSION\_STATE  
 join ...  
 WHERE ((sru\_id = 5) and (P1\_VALID\_FROM is NULL))  
 or ((sru\_id = 6) and (P1\_VALID\_FROM >= P1\_VALID\_TIL))  
 or ((sru\_id = 7) and (P1\_VALID\_FROM is not NULL))

Depending on the application state, different rules are selected whose conditions currently evaluate to TRUE. The query will, based on the sort criterion of the rules (SORT\_SEQ column of the ADC\_RULE table) select and apply the rule with the lowest sort. Based on the selected rule, the actions are then available in PL/SQL and JavaScript via the connection of the ADC\_RULE, ADC\_RULE\_ACTION and ADC\_ACTION\_TYPE tables and can be combined by the code into one response.

All available rules are made available to the rule view by a view ADC\_BL\_RULES, which provides a list of all available rules and their activities from the individual base tables. To increase query performance, this view is pre-filtered using the CGR\_ID column. By joining the ADC\_BL\_RULES view to the rule actions, multiple rows are returned. To ensure that all rows of the selected rule are recognized, a ranking is created via the CRU\_ID column and only the rows that are ranked 1 are determined.

#### Recursive rule execution

Rule groups are set by default so that rules are executed recursively. This can also be suppressed by setting a flag for the rule group. In this case, only the rule that has the highest validity for the current application state situation (i.e. the first rule whose condition evaluates to TRUE) is always and exclusively executed.

If the application state of an element is changed in a rule, this is registered in the plugin, as already described, if the PLUGIN\_ADC.SET\_SESSION\_STATE method is used. This initially results in the changed value being reported back to the application, where it is updated in the page element.

An extension to this behavior exists, which is always taken into account when the element ...

* is not identical with the triggering element and
* the element is a "relevant" element, i.e. rules of the rule group refer to this element.

In this case, after processing the original event but before creating a response, the plugin will simulate a CHANGE event for each element in question and request a corresponding response from the plugin. By this mechanism it is possible to assign values to several elements in one action and - before the answer to the calling program - to check all changed elements, to which a rule refers, in order to determine whether these elements should be displayed or not, for example. All database actions resulting from these checks are executed, and all JavaScript actions are collected in a response. To distinguish which JavaScript statement was generated by which rule, the response contains a comment entry before each new call with information about the rule and the recursion depth that generated this JavaScript statement.

Since this mechanism can cause infinite loops, a recursion counter is updated in the plugin, which stops further processing at a maximum of 10 recursions. The recursion is not called again for an element that has already been processed, regardless of the recursion depth at which this new call occurs.

As an application example for this recursive rule execution, let us assume the following.:

An input field on the form allows to enter a customer ID. By changing the customer ID, the plugin should fill several fields of the customer with data. If the customer is a commercial customer, the P1\_COMPANY field shall be displayed, otherwise this element shall be hidden.

A rule is created for the P1\_CUSTOMER\_ID field, stating that it should be executed if the field is not empty. The action should be to call a PL/SQL method that assigns values of this customer to several elements in the application state. Two other rules state that if the P1\_CUSTOMER\_TYPE field contains the value C, the P1\_COMPANY field should be shown, otherwise not.

If the P1\_CUSTOMER\_TYPE field is now set by the PL/SQL method, without recursive rule handling the CHANGE event for the element would not be triggered and the associated rule would not be considered. Recursive rule execution allows the plugin to recognize that the P1\_CUSTOMER\_TYPE field has been changed and that a rule relates to this field. This re-executes the rule processing for this element as the triggering element and adds the JavaScript action to the response. Since no other elements were changed by this rule, no further recursive activity is triggered and the response is sent to the calling page.

The following listing shows a typical response for a relatively deeply nested page. Here the scenario is that five selection lists have to be filled in succession in order to copy a rule group to another application. Initially, if the top selection list is unoccupied, all other selection lists should be disabled. Without recursion, the rules for the first selection list would have to define the state of all other selection lists, as well as the following one for all others, etc. With recursion, this is not necessary: The first selection list deactivates the second, which in turn deactivates the third, and so on. With recursion the following (formatted) answer results:

<script>

de.condes.plugin.ADC.setItemValues(

[{"id":"P4\_SGR\_PAGE\_ID","value":""},

{"id":"P4\_SGR\_ID","value":""},

{"id":"P4\_SGR\_APP\_TO","value":""},

{"id":"P4\_SGR\_PAGE\_TO","value":""}]);

de.condes.plugin.ADC.setErrors(

{ "count":0,

"errorDependentButtons":"B4\_COPY",

"firingItems":"P4\_SGR\_PAGE\_ID,P4\_SGR\_ID,P4\_SGR\_APP\_TO,P4\_SGR\_PAGE\_TO",

"errors":[]});

//Recursion 1: RULE\_10 (Quellanwendung ist leer), Firing Item: DOCUMENT

apex.item('P4\_SGR\_PAGE\_ID').show;

apex.item('P4\_SGR\_PAGE\_ID').disable();

//Recursion 2: RULE\_30 (Quellseite ist leer), Firing Item: P4\_SGR\_PAGE\_ID

apex.item('P4\_SGR\_ID').show;

apex.item('P4\_SGR\_ID').disable();

//Recursion 3: RULE\_50 (Regelgruppe ist leer), Firing Item: P4\_SGR\_ID

apex.item('P4\_SGR\_APP\_TO').show;

apex.item('P4\_SGR\_APP\_TO').disable();

//Recursion 4: RULE\_70 (Zielanwendung ist leer), Firing Item: P4\_SGR\_APP\_TO

apex.item('P4\_SGR\_PAGE\_TO').show;

apex.item('P4\_SGR\_PAGE\_TO').disable();

//Recursion 5: RULE\_90 (Zielseite ist leer), Firing Item: P4\_SGR\_PAGE\_TO

apex.item('B4\_COPY').show();

apex.item('B4\_COPY').disable();

</script>

Recursive rules allow to simplify rule sets. Here is an example: Application elements are to be initialized on a page: Some buttons and input elements are to remain disabled as long as an ID is not set on the page. One will now - without recursive rule execution - start defining the state of all buttons. With recursive rule execution, it is enough to set the ID element to NULL and define a rule that specifies which buttons should be hidden if this element does not contain a value. No matter by what means this element is now set to NULL, this rule will recursively always be called and perform the corresponding action.

Recursion makes the rules more modular and therefore easier to reuse. In the example above, only individual rules have been defined for the various elements. Depending on the triggering element, the respective dependent elements are now called recursively and the status is always set correctly.

#### Initialization rules

Rules can be provided with an "Execute on page load" flag. By default, this flag is FALSE, which means that the rule can only be executed if the triggering element was addressed in the rule condition. If the flag is set, the rule will be executed in addition to the generic initialization rule if the technical condition evaluates to TRUE and the page is initialized.

This option can be used to reduce the number of rules and roughly corresponds to the corresponding option of Dynamic Actions. A rule executed by this flag is always executed in addition to, and after, the initialization rules (initializing = C\_TRUE) that are present anyway. If the flag is set for multiple rules, multiple rules will be executed, even if recursion has been prohibited for the rule group. The initialization rules are not executed by the mechanism of recursion, but in parallel, based on the setting of this flag. Initialization rules do not result in recursion, even if it is allowed for the rule group. This prevents recursion loops.

## Technical implementation: modules

The plugin consists of several modules:

* Core functionality  
  Implemented in the packages ADC\_INTERNAL and ADC\_API and the tables ADC\_%
* Plugin  
  The actual APEX plugin, implemented in the package PLUGIN\_ADC and the JavaScript files adc.js and adcApex.js
* PL/SQL API  
  Used to determine actions by PL/SQL codes. This is a way to implement very complex sets of rules. The API is stored in the package ADC.
* Administration package  
  Used to export rule groups, but also to create and validate Decision Table logic. It is implemented in the package ADC\_ADMIN.
* APEX Application  
  APEX application for managing rule groups, rules, actions and action types, implemented in the package ADC\_UI as well as the APEX application itself.

### Core functionality: Packages ADC\_API and ADC\_INTERNAL

The package ADC\_INTERNAL contains the central logic of ADC and is not released for direct use. The methods that may be used externally are published in ADC\_API. These methods are called by the packages PLUGIN\_ADC, ADC as well as ADC\_UI. ADC\_INTERNAL relies in parts on ADC\_ADMIN, e.g. to create the decision table logic. This is also needed when importing rule groups.

Currently the following behavior is implemented:

* When a rule group or rule is created or modified, an SQL query is created, as explained in the example, and stored as an SQL query in table ADC\_RULE\_GROUPS.
* When an action is created, the SQL statement is executed, the resulting rule is determined, and the response is generated from the metadata of this rule and passed to the calling code
* If a rule group is exported, the metadata is transferred into calls of the administration packages and these are transmitted as CLOB to the calling code
* Method of analyzing a rule set and creating the PL/SQL and JavaScript scripts in response of the plugin to a given application situation, providing auxiliary functions for the packages. PLUGIN\_ADC and ADC\_UI

Here is a short overview of some relevant functions:

#### Getter methods GET\_FIRING\_ITEM, GET\_JS\_FUNCTION

The methods return the name of the triggering element and the name of the JavaScript function used to call the plugin. They are called by the rule view and the PLUGIN\_ADC package.

#### Method CREATE\_ACTION

This method calculates which rule is relevant for a given situation in the application state, as well as the resulting actions for the database and the application. From the determined rule, the method calculates an anonymous PL/SQL block with the PL/SQL statements as well as a list of JavaScript functions for the application. The execution of the PL/SQL code and the preparation of the response for the APEX application are done in the package PLUGIN\_ADC.

The method accesses the values of the application state and all stored rules by evaluating the rule view created by the package when maintaining the rules using an SQL query. This query determines all valid rules and links them to all PL/SQL and JavaScript actions via the ADC\_ACTION\_TYPE table. From the determined result set, the first rule that evaluated to TRUE is selected and the actions referenced in it are combined in a PL/SQL and a JavaScript block. In addition, all elements are determined that are related to the triggering elements in a rule context. The list of these elements is passed as a comma-separated list to the calling code, which derives from it for which elements error messages are to be removed from the interface.

#### Method SET\_SESSION\_STATE

For the correct function of the plugin it is important to set element values in the application state exclusively via the method SET\_SESSION\_STATE and not directly via APEX\_UTL.SET\_SESSION\_STATE to make sure that the plugin contains knowledge of the value change and can pass it on to the application.

All elements that are changed by processing the rule request are registered by the plugin and passed to the application page in the response with their current element values from the application state. The plugin sets the element values so that the page and application state are subsequently synchronized with respect to the changed elements.

#### Method REGISTER\_ERROR

The REGISTER\_ERROR method collects all errors that occur during processing and passes them on to the JavaScript methods of the plug-in so that they can be displayed there as errors for the corresponding elements. An error is defined by the name (ID) of the application element and the error text. By specifying the ID of the application element that triggered the error, the plugin is able to display the error at the affected element and selectively delete it from the application page.

#### Method REGISTER\_MANDATORY

The method notes for an element that it represents a mandatory element. The status can be changed by rules, so that it is possible to declare elements dynamically as mandatory elements or to withdraw this status. All mandatory page items are maintained per session in an APEX collection to allow for different sets of mandatory page items per APEX session.

#### Method CHECK\_MANDATORY

The method checks all mandatory fields managed by this rule group and outputs error messages for each element that does not contain a value.

#### Method REGISTER\_NOTIFICATION

Method REGISTER\_NOTIFICATION, analogous to method REGISTER\_ERROR, collects notifications to the application user that have accumulated during processing of the rule. After processing is completed, all rules will be summarized and displayed in a notification block on the APEX page. The notifications may be confirmation messages, status messages or similar.

#### Method PROCESS\_RULE

This private method is the core method of the plugin. It executes the rule logic and controls the recursive invocation of the rules. The operation of the method is organized as follows:

* Before calling the method, the error stack and the list of recursively required calls, the recursion stack, were cleaned up. The only element in the list of recursive calls is the triggering element, the recursion depth counter is set to 1.
* The method notes the current state of the recursion depth and increments this value at the beginning. If further elements have to be processed recursively, these would be noted in the recursion stack with their name and the incremented recursion depth. If, for example, element P1\_ID had to be recursively processed, this element would be noted in the list as entry P1\_ID with the value 2.
* Now the method iterates over all elements of the recursion stack and searches for elements whose recursion depth corresponds to the current state of the recursion depth (1 at the beginning). If this is successful, an edit flag is set to indicate that another recursive call to the method will be required. If this flag is not set, the recursion is aborted and the method terminates. Before the method is called recursively, all elements of the current recursion level are processed.
* If the method has found an entry in the recursion stack, this is considered the triggering element and the rule logic is evaluated for this element.
* The PROCESS\_RULE method itself does not contain any decision logic and cannot evaluate rules. The method delegates this task to ADC\_ADMIN.CREATE\_ACTION and from there contains information about which actions are to be executed in PL/SQL within the database and in JavaScript in the browser. Once the rule set is evaluated, a PL/SQL script and a JavaScript script are delivered to the package. In addition, a list of triggering elements (FIRING\_ITEMS) is noted. This refers to the elements that are affected by the evaluated rules. This leads to the fact that error messages, which could be on the surface, are removed from these elements and - if present - are set again. This prevents the removal of error messages on the surface that are not affected by the current rules.
* The PROCESS\_RULE method executes the PL/SQL code determined by the rule evaluation in ADC\_ADMIN.CREATE\_ACTION. If as part of the execution of the PL/SQL block the method SET\_SESSION\_STATE of the plugin was called and thus an element value in the application state was changed, this element is marked for recursive processing: The SET\_SESSION\_STATE method calls the REGISTER\_ITEM private method as part of its execution, which inserts the changed element into the recursion stack if the changed element is not yet there and the recursion depth has not yet exceeded the maximum recursion depth specified by the C\_RECURSIVE\_LIMIT package variable.
* The currently processed element is removed from the recursion stack after processing is complete. Since the processing flag was set when the element was processed to indicate that elements were processed at this recursion level, the PROCESS\_RULE method is now called recursively. The process starts over, but this time with the recursion depth incremented.

After completing all recursive rule evaluations, the method returns a cumulative JavaScript statement that is sent by the plugin to the interface and executed there. The PL/SQL statements have already been executed as part of the rule evaluation and are thus complete.

The way the recursion works, which first processes all elements of a recursion depth and then processes elements of the next level (breadth first), ensures that the subsequent recursion depth can access the modified application state of the previous recursion depth: Elements set at recursion depth 1 are available in recursion depth 2 in the rule view.

#### Method REGISTER\_ITEM

The REGISTER\_ITEM method is a private method that is called when a value in the application state is changed by the PLUGIN\_ADC.SET\_SESSION\_STATE method. The method gets the element name of the element that was changed and tries to find it in the list of relevant elements of the rule group. If the calling element itself changes (i.e. if the element name is the same as the current FIRING\_ITEM), the element is ignored to prevent infinite loops.

If the changed element is relevant and not equal to the triggering element, an attempt is made to write the element to the recursion stack under the currently valid recursion depth. This is rejected by the stack if the changed element is already there. In this case, the plugin returns the error message 'Element has recursive loop created and therefore ignored.' for the element. Based on a parameter setting it is possible to raise this message as an exception or to silently ignore it.

The recursion stack also rejects the inclusion of the element if a maximum recursion depth has been exceeded. The maximum recursion depth is currently specified by a package constant C\_RECURSIVE\_LIMIT. Possibly this constant should be outsourced as a parameter. If this limit is exceeded, the error message 'Element has exceeded recursion depth of <n>.' is returned for the current element.

### Administration: Package ADC\_ADMIN

The package includes several functional areas:

* Administration and master data maintenance  
  Creation and management of rule groups, rules, actions and action types, methods for importing and exporting rule groups to integrate the plugin into an APEX deployment cycle.
* Internal administration  
  Automated creation of rule queries, checking of rules

On the one hand, the metadata of the plugin are directly maintained by wizard-based pages of the APEX application, on the other hand, there is an API for creating rules, for example, from an export of the rule group or rule groups. They are not used by APEX application but serve for import of rule groups into an application. The methods of the package available for this purpose represent simple wrappers around MERGE statements, with the help of which the master data tables can be maintained.

Methods that work in this way are:

* MERGE\_RULE\_GROUP
* MERGE\_RULE
* MERGE\_RULE\_ACTION
* MERGE\_ACTION\_TYPE
* DELETE\_RULE\_GROUP
* RESEQUENCE\_RULE\_GROUP

In addition, the package provides auxiliary methods for deleting rule groups, rules, etc.

#### Methods EXPORT\_RULE\_GROUP and EXPORT\_RULE\_GROUPS

EXPORT\_RULE\_GROUP allows exporting a rule group by ID. Method EXPORT\_RULE\_GROUPS enables export of all rule groups of an application. The methods return the export scripts as CLOB, the application has a maintenance page that can be used to run the exports and load them as a file.

When exporting a rule group, it can be specified whether the export is to be done for another application and/or application page. Thus, the export function can be used to copy rules between applications or to integrate rules into the script for installation when exporting an APEX application. If the rule group for the requested application ID and page ID exists, the export will overwrite the existing rule group.

#### Method DELETE\_RULE\_GROUP

The method removes the specified rule group from the master data tables.

#### Method PROPAGATE\_RULE\_CHANGE

The PROPAGATE\_RULE\_CHANGE method is called after each change to a rule or rule group. The call of the method causes that...

* The rule is validated
* The list of relevant elements of the control group is recalculated
* A new version of the decision rule logic is created

The method is called by the APEX application to maintain the plugin after each change. The APEX application is based on wizard-create pages for master data maintenance, therefore the standard processes for creating and saving the data are also used. To avoid a trigger that triggers the validation of the rule change, on the relevant pages of the APEX application this method is explicitly called after the standard processes of the page.

### Plugin: Package PLUGIN\_ADC

The package PLUGIN\_ADC implements the interface for a dynamic action plugin as specified by APEX, namely the methods RENDER and AJAX.

#### Method RENDER

The RENDER method of the package initializes the plugin by creating a list of relevant form elements for a rule group, to which the package then binds event handlers. CHANGE events can be bound to form elements and CLICK events to form buttons. As already described, the selection of the elements to which the plugin binds is done automatically via an analysis of the defined rules.

The sequence of the rendering process is as follows:

* The plugin is created, based on the parameter values of the plugin on the application page, and set up on the page
* The database determines all relevant elements of the rule group and passes them as parameters to the plugin
* The plugin binds corresponding event handlers to the events stored from the database for the corresponding element type (form elements bind to the CHANGE event, buttons bind to the CLICK event, see table ADC\_PAGE\_ITEM\_TYPE)
* At the end of the initialization, the plugin calls the ADC\_INTERNAL.PROCESS\_REQUEST function for the first time and passes
  + The current element value of all relevant elements
  + The DOCUMENT specification as the triggering item.

Then proceed with this call as described in the next section.

#### Method AJAX

The AJAX method implements the behavior of the plugin when a bound event is fired on the application page.

The JavaScript functionality of the plugin compiles the current element values of the relevant elements and persists them in the application state and passes the name of the triggering element to the AJAX method. After the plugin is initialized on page load, this function is called for the first time with the triggering element DOCUMENT to ensure that the application page is initialized correctly.

In the AJAX method, the following steps are initiated:

* An internal error stack is initialized.  
  The incoming error messages are stored on this error stack by the REGISTER\_ERROR method.
* A recursion stack is initialized.  
  The recursion stack is a PL/SQL table that uses the ID of the triggering element as key values and the recursion depth starting at 1 as load
* The rule group is processed.  
  The processing of the method is done by ADC\_INTERNAL.PROCESS\_RULE, the description of the individual processing steps can be found at its documentation.
* Once the rule processing is completed, the changed elements and the collected JavaScript instructions will be integrated into a JSON-formatted response and sent to the APEX application page. Error messages are integrated as well as a list of the elements that were affected by the specific rule. This list is used to remove error messages present on the page.
* The plugin executes the instructions of the page and thus completes the editing cycle.

### Plugin: JavaScript file adc.js

The JavaScript file adc.js implements the JavaScript-side logic of the ADC plugin. The full implementation is split between two JavaScript files. File adc.js takes care of the plugin's logic, but excludes APEX display-specific logic that depends on the StyleSheet, template, etc. that may be used. This display logic is implemented in file adcAPEX.js.

The adc.js file implements the namespace de.condes.plugin.adc (hereafter: ADC) and declares a selection of methods in this namespace that serve to operate the plugin on the application side:

#### Method INIT

This method is used to initialize the plugin. The database sends a data structure during page load, which is received and evaluated by this method. The statement of the database contains two attributes (besides the AJAX-Identifier, which has to be generated for all APEX plugins according to default):

1. A JSON object with the list of "relevant" page elements. Each entry contains information about the ID of the element and the event to be bound by the plugin.
2. A comma-separated list of page elements to be sent to the database by the plugin. During the initialization phase all relevant elements and possibly other elements are included in this list, during the further calls the list is generated from the list of changed page elements.

As APEX masks attributes sent to a plugin for security reasons and a masked parameter, on the other hand, cannot be interpreted as JSON object by JavaScript without further ado, control characters will be masked by database and unmasked by plugin before being sent. Only then is the response parsed as a JSON object. The list of page elements is transferred to a global storage structure of the plugin as an array.

Based on the transmitted JSON object with the page elements to be bound, the corresponding event handlers are then set up on these elements and the plugin is initially executed. This is done in the EXECUTE method.

#### Method EXECUTE

The method is called when a bound page element has thrown an event, or when the plugin is initialized. In this case, DOCUMENT is passed as the triggering element.

This method implements the core functionality of the plugin's JavaScript page. The flow is to collect all "relevant" page items from ADC.bindItems and pass them to an AJAX call that computes a response, based on the rules, for the current application state situation. This is achieved by a given option of APEX functionality apex.server.plugin: when creating an AJAX request to the database, there is a parameter pageItems which can be passed a list of page items whose values will be written to application state automatically by this method. Another parameter passed to this method is the ID of the triggering page item (or DOCUMENT).

This data can be seen when the POST part is parsed for the AJAX call.

As usual with AJAX, the request to the database is asynchronous. To enable this, a callback method is provided, which is called as soon as the database response arrives.

The database returns an HTML fragment with embedded JavaScript as a response. This HTML fragment is inserted into the document by the callback method. The insertion is done by the JQuery method $.append, which has the property of directly executing JavaScript statements embedded in HTML. Since this completes the task of the HTML fragment, it can be removed from the document immediately afterwards.

The HTML fragment contains method calls for the plugin and passes corresponding JSON objects that are evaluated in the individual methods. The following listing shows an example database response to the AJAX call:

<script>

de.condes.plugin.ADC.setItemValues(

[{"id":"P4\_SGR\_PAGE\_ID","value":""},

{"id":"P4\_SGR\_ID","value":""},

{"id":"P4\_SGR\_APP\_TO","value":""},

{"id":"P4\_SGR\_PAGE\_TO","value":""}]);

de.condes.plugin.ADC.setErrors(

{ "count":0,

"firingItems":"P4\_SGR\_PAGE\_ID,P4\_SGR\_ID,P4\_SGR\_APP\_TO,P4\_SGR\_PAGE\_TO",

"errors":[]});

//Recursion 1: RULE\_10 (Quellanwendung ist leer), Firing Item: DOCUMENT

apex.item('P4\_SGR\_PAGE\_ID').show;

apex.item('P4\_SGR\_PAGE\_ID').disable();

</script>

Three areas can be identified:

1. A call to the method ADC.setItemValues  
   This method is used to update the passed element values from the application state, which were changed within the database as part of processing the request, on the application side.
2. A call to the method ADC.setErrors  
   This method contains a JSON object with information about the errors that occurred during processing. First, the errors of all the page elements that are noted in the firingItems entry are removed. Then, the items that were inserted in the errors array receive an error message.
3. The last section of the response contains the JavaScript script that was computed as the action for the selected rule. This entry may have been called and compiled recursively. Therefore, this part can be quite extensive. To facilitate error analysis, the recursion depth and the rule used for this recursion as well as the triggering element are noted for all subscripts that were created by a recursion.

#### Auxiliary methods

The plugin has further auxiliary methods that take over partial tasks. These are for the most part trivial and will be discussed summarily here:

* ADC.setItemValues  
  Helper method that iterates over a JSON object with page element IDs and values and updates the affected page elements.
* ADC.bindEvents  
  The method is passed a JSON object with the relevant page elements and the events to bind to. The method iterates over this object and binds the relevant events.  
  A special feature of this method is that the events of elements that are updated by an APEX event apexrefresh are unbound from the event handler before this event is executed and bound again after the event is completed. This avoids that bound events may be triggered by an APEX activity and thus may cause endless loops.
* ADC.setErrors  
  Helper method that updates the error status of the application page. Since this depends on the theme used, the method is just a wrapper around the corresponding method from adcAPEX.js.
* ADC.setMandatory  
  If an element is declared as mandatory by ADC, this must be indicated by adjustments to the representation on the interface. This is done by this method, which is also a wrapper around adcAPEX.js.
* ADC.submit  
  This helper method also represents a wrapper around a corresponding method of the adcAPEX.js file. The task of the method is to check whether the page still displays an error and if so, to output a corresponding message. If there is no error on the page, the page is submitted using the apex.submit method. A value for the REQUEST variable can be passed to the method as a parameter.
* de\_condes\_plugin\_adc  
  This method is not part of the JavaScript namespace. It serves for calling method adc.init. For reasons that are not entirely clear, APEX apparently has difficulty calling a namespace method directly. So this method is a workaround and might become obsolete in the future if a better way to address the plugin is found.

### Plugin: JavaScript File adcAPEX.js

That JavaScript file implements visual representation of functionalities requested by plug-in. Reason for splitting into two is that functionality is implemented in this file which depends on concrete APEX version, theme used or customer-specific extension thereof. Encapsulation in a separate file serves for better separation of responsibilities: The file adc.js is in principle independent of the presentation.

A core functionality of the file is the representation of errors generated by the plug-in. APEX has a mechanism for rendering errors, but it is not powerful enough because, for example, removing targeted messages is not supported. Therefore, functionality had to be programmed in order to achieve consistent representation of errors, no matter whether they have been generated by APEX or by plugin ADC.

The individual methods are largely self-explanatory as far as their field of application is concerned. In detail, however, some complex JQuery operations are required to achieve the desired visual appearance. However, since this depends on the concrete design environment, this will not be discussed in detail here. It remains with a summary description:

* Method <Namespace>.setErrors  
  The method creates error messages on the page. Unlike APEX-generated errors, plugin-generated error messages in the notification area always include a CSS class corresponding to the ID of the element to which the message refers. This allows error messages in the notification area to be selectively removed.  
  In addition, the error messages in the notification area are accompanied by links that can be used to branch to the corresponding location in the code.

### APEX Administration Application

The APEX application of the plug-in is used for simplified maintenance of rule groups, rules, actions and action types. In addition to the basic functions, i.e. creation, maintenance and deletion of the described elements, the application has a validation function that validates rules against the data dictionary of the APEX application for which the rule is created.

The application itself is built using standard APEX means, and the master data tables are maintained using a transaction API. In addition, after changes are saved, the application propagates the changes, i.e. it initiates the validation as well as the recreation of the decision tables.

The application uses the plugin to show buttons, update reports and more. The rules needed for this are hidden in the rule groups overview to avoid accidental deletion of these rules.

### APEX Application: Package ADC\_UI

Package ADC\_UI implements application logic for APEX application for administration of rule groups, rules, actions and action types. The package delegates the actual logic to ADC\_API and ADC\_ADMIN. Therefore the methods of the package are merely wrappers around the methods of the package ADC\_API and ADC\_ADMIN, partly with different parameter equipment.