# IRS BPM DECISION SYSTEMS FLOW DESIGN DOCUMENT

## IRS BPM Analytics Model

IRS BPM Demo Decision Systems Flow

Master

File

RUP

CADE

RRP

MeF

IDRS

EUP

AUR

AIR

CFOL

* (RUP) Registered User Portal
* (EUP) Employee User Portal
* (MeF) Modernized e-File
* (AIR) Act Information Return
* (CADE) Customer Account Data Engine
* (MFOL) Master File On–Line
* (IDRS) Integrated Data Retrieval System
* (CFOL) Corporate Files on Line
* (RRP) Return Review Program
* (AUR) Automated Under-reporter

IRS BPM Decision System - Real time analytics and correlation system, ability to stich multiple events and data sources, to provide system nodes service level view for multiple subscribers.

* Complex Event Processing
* Rule based correlation
* BPM based process flows
* Rule management.

Included in this system would be the capability for each failure type:

* Failure condition
* Failure logic: Pass/Fail or Threshold
* Information Source
* Currently captured or not
* Potential behavior and impacted functions upon failure

## Solution Design Considerations

### Design Methodology

Following key design methodologies are utilized in the IRS BPM Decision Systems application:

* **Business Decision Engine**

Provide a mechanism to define, deploy, execute, monitor and maintain the variety and complexity of decision logic that is used by operational systems within an organization. Most rules used by businesses decision systems are forward chaining, which can be further divided into two classes:

* + The first class processes so-called production/inference rules. These types of rules are used to represent behaviors of the type: WHEN condition THEN action. For example, such a rule could answer the question: "Should the health status of this node be ALERT/ERROR?" by executing rules of the form "WHEN some-condition THEN change-health-status-to-ALERT/ERROR".
  + The other type of rule engine processes so-called reaction/Event Condition Action rules. The reactive rule engines detect and react to incoming events and process event patterns. For example, a reactive rule engine could be used to alert when certain errors happen 5 times in 1 minute.
* **The IRS BPM Error Condition Rules**

Most of the Rules used by IRS BPM Decision System are Event Declaration; to declare a fact type as an “event”, all it is required is to assign the @role metadata tag to the fact type.

The @role metadata tag accepts two possible values:

* **Fact**: this is the default, declares that the type is to be handled as a regular fact.
* **Event**: declares that the type is to be handled as an event. Every event has an associated timestamp assigned to it.

By default, the timestamp for a given event is read from the Session Clock and assigned to the event at the time the event is inserted into the working memory. All facts are static and stored in the IRS BPM Decision System Production Memory as Rules or Events. The facts that the Inference Engine matches against are kept in the Working Memory. Using the CEP pattern of processing the Event Driven Architecture from the alerting system, the facts are inserted into the IRS BPM Decision Systems Working Memory when the Data Ingestion occurs.

* Defining terms is not the goal of this guide and as so, let adopt a loose definition that, although not formal, will allow us to proceed with a common understanding. So, in the scope of this guide: Event is a record of a significant change of state in the application domain at a given point in time. Events are immutable and can be embellished.
* **Event-Driven Architectures**

An event-driven architecture (EDA) is a software architecture pattern promoting the production, detection, consumption of and reaction to events. Building applications and systems around an event-driven architecture allows these applications and systems to be constructed in a manner that facilitates more responsiveness, because event-driven systems are, by design, more normalized to unpredictable and asynchronous environments.

Event processing is a method of tracking and analyzing (processing) streams of information (data) about things that happen (events), and deriving a conclusion from them. Complex Event Processing, or CEP, is event processing that combines data from multiple sources to infer events or patterns that suggest more complicated circumstances. The goal of complex event processing is to identify meaningful events (such as error conditions) and respond to them as quickly as possible.

These events are notices happening across the various layers of the organization. An event may also be defined as a "change of state," when a measurement exceeds a predefined threshold of time, response, or other value. IRS BPM Decision Systems CEP Analysts will give the organizations a new way to analyze patterns in real-time, and help the business side communicate better with IT and service departments.

After inserted new events into Working Memory from the Data Collection ingestion the Event Stream Processing, or ESP, is a set of technologies which include event visualization, event databases, event-driven middleware and our event processing language rules. After inserting events, the ESP passes-off to our complex event processing (CEP), which takes president.

**CEP - Complex Event Processing**

CEP is primarily an event processing concept that deals with the task of processing multiple events with the goal of identifying the meaningful events within the event cloud. CEP employs techniques such as detection of complex patterns of many events, event correlation and abstraction, event hierarchies, and relationships between events such as causality, membership, and timing, and event-driven processes.

CEP deals with the task of processing streams of event data with the goal of identifying the meaningful pattern within those streams, employing techniques such as detection of relationships between multiple events, event correlation, event hierarchies, and other aspects such as causality, consequence and timing. These are inserted in our BPM engine and synthetic facts or events that are conclusions of facts.

CEP allows patterns of simple and ordinary events to be considered to infer that a complex event has occurred. Complex event processing evaluates a confluence of events and then takes action. The events (notable or ordinary) may cross event types and occur over a long period of time. The event correlation may be causal, temporal, or spatial. CEP requires the employment of sophisticated event interpreters, event pattern definition and matching and correlation techniques. CEP is commonly used to detect and respond to business anomalies, threats, and opportunities and is well suited for Service Assurance domains.

* 1. **BPM Fusion Solution Design**
     1. **Design Use Case Methodology**

Event Processing use cases, share several requirements and goals with Business Rules use cases.

These overlaps happen both on the business side and on the technical side.

Supporting Complex Event Processing, though, is much more than simply understanding what an event is. CEP scenarios share several common and distinguishing characteristics:

* Usually required to process huge volumes of events, but only a small percentage of the events are of real interest.
* Events are usually immutable, since they are a record of state change.
* Usually the rules and queries on events must run in reactive modes, i.e., react to the detection of event patterns.
* Usually there are strong temporal relationships between related events.
* Individual events are important. The system is concerned about patterns of related events and their relationships.
* Usually, the system is required to perform composition and aggregation of events.

Based on this general common characteristic, BPM Fusion defined a set of goals to be achieved in order to support Complex Event Processing appropriately:

* Support Events, with their proper semantics, as first class citizens.
* Allow detection, correlation, aggregation and composition of events.
* Support processing of Streams of events.
* Support temporal constraints in order to model the temporal relationships between events.
* Support sliding windows of interesting events.
* Support the required volumes of events for CEP use cases.
* Support to (re)active rules.
* Support adapters for event input into the engine (pipeline).

Additionally, in some scenarios, you will have to discard equal objects (objects of the same type and values) when they are inserted into the working memory, to avoid data inconsistency and unnecessary activations.

A system of streams where the events are transmitted, and these events can be of the same type but have to be processed in different ways without mixing them in their processing. BPM Fusion can handle this scenario by creating entryPoints that can be used to integrate these streams with the rules patterns, to process the events that are going to arrive from these streams.

* 1. **Event Semantics**

An *event* is a fact that presents a few distinguishing characteristics:

* **Usually immutables:** since, by the previously discussed definition, events are a record of a state change in the application domain, i.e., a record of something that already happened, and the past cannot be "changed", events are immutables. This constraint is an important requirement for the development of several optimizations and for the specification of the event lifecycle, and one of the most common use cases for rules is event data enrichment.
  + **Note** As a best practice, the application is allowed to populate un-populated event attributes (to enrich the event with inferred data), but already populated attributes should never be changed.
* **Strong temporal constraints:** rules involving events usually require the correlation of multiple events, especially temporal correlations where events are said to happen at some point in time relative to other events.
* **Managed lifecycle:** due to their immutable nature and the temporal constraints, events usually will only match other events and facts during a limited window of time, making it possible for the engine to manage the lifecycle of the events automatically. In other words, one an event is inserted into the working memory, it is possible for the engine to find out when an event can no longer match other facts and automatically delete it, releasing its associated resources.
* **Use of sliding windows:** since all events have timestamps associated to them, it is possible to define and use sliding windows over them, allowing the creation of rules on aggregations of values over a period of time. Example: average of an event value over 60 minutes.

BPM supports the declaration and usage of events with both semantics: **point-in-time** events and **interval-based** events.

* + **Note** A simplistic way to understand the unification of the semantics is to consider a *point-in-time* event as an *interval-based* event whose *duration is zero*.
  1. **Event Processing Modes**

Rules engines in general have a well-known way of processing data and rules and provide the application with the results. Also, there are not many requirements on how facts should be presented to the rules engine, especially because in general, the processing itself is time independent. That is a good assumption for most scenarios, but not for all of them. When the requirements include the processing of real time or near real time events, time becomes and important variable of the reasoning process.

The following sections will explain the impact of time on rules reasoning and the two modes provided by BPM for the reasoning process.

* + 1. **Cloud Mode**

The CLOUD processing mode is the default processing mode. Users of rules engine are familiar with this mode because it behaves in exactly the same way as any pure forward chaining rules engine, including previous versions of BPM.

When running in CLOUD mode, the engine sees all facts in the working memory, does not matter if they are regular facts or events, as a whole. There is no notion of flow of time, although events have a timestamp as usual. In other words, although the engine knows that a given event was created, for instance, on January 1st 2009, at 09:35:40.767, it is not possible for the engine to determine how "old" the event is, because there is no concept of "now".

In this mode, the engine will apply its usual many-to-many pattern matching algorithm, using the rules constraints to find the matching tuples, activate and fire rules as usual.

This mode does not impose any kind of additional requirements on facts. So for instance:

* There is no notion of time. No requirements clock synchronization.
* There is no requirement on event ordering. The engine looks at the events as an unordered cloud against which the engine tries to match rules.

On the other hand, since there are no requirements, some benefits are not available either. For instance, in CLOUD mode, it is not possible to use sliding windows, because sliding windows are based on the concept of "now" and there is no concept of "now" in CLOUD mode.

Since there is no ordering requirement on events, it is not possible for the engine to determine when events can no longer match and as so, there is no automatic life-cycle management for events. i.e., the application must explicitly delete events when they are no longer necessary, in the same way the application does with regular facts.

* + 1. **Stream Mode**

The STREAM processing mode is the mode of choice when the application needs to process streams of events. It adds a few common requirements to the regular processing, but enables a whole lot of features that make stream event processing a lot simpler.

The main requirements to use STREAM mode are:

* Events in each stream must be time-ordered. i.e., inside a given stream, events that happened first must be inserted first into the engine.
* The engine will force synchronization between streams through the use of the session clock, so, although the application does not need to enforce time ordering between streams, the use of non-time-synchronized streams may result in some unexpected results.

When using the STREAM, the engine knows the concept of flow of time and the concept of "now", i.e., the engine understands how old events are based on the current timestamp read from the Session Clock. This characteristic allows the engine to provide the following additional features to the application:

* Sliding Window support
* Automatic Event Lifecycle Management
* Automatic Rule Delaying when using Negative Patterns
  + - 1. **Role of Session Clock in Stream mode**

When running the engine in CLOUD mode, the session clock is used only to time stamp the arriving events that don't have a previously defined timestamp attribute. Although in STREAM mode, the Session Clock assumes an even more important role.

In STREAM mode, the session clock is responsible for keeping the current timestamp, and based on it, the engine does all the temporal calculations on event's aging, synchronizes streams from multiple sources, schedules future tasks and so on.The documentation on the Session Clock section to know how to configure and use different session clock implementations.

* + - 1. **Negative Patterns in Stream Mode**

Negative patterns behave different in STREAM mode when compared to CLOUD mode. In CLOUD mode, the engine assumes that all facts and events are known in advance (there is no concept of flow of time) and so, negative patterns are evaluated immediately.

When running in STREAM mode, negative patterns with temporal constraints may require the engine to wait for a time period before activating a rule. The time period is automatically calculated by the engine in a way that the user does not need to use any tricks to achieve the desired result.

* 1. **Streams Support**

Most CEP use cases have to deal with streams of events. The streams can be provided to the application in various forms, from JMS queues to flat text files, from database tables to raw sockets or even through web service calls. In any case, the streams share a common set of characteristics:

* **events** in the stream are ordered by a timestamp. The timestamp may have different semantics for different streams but they are always ordered internally.
* **volumes** of events are usually high.
* **atomic events** are rarely useful by themselves. Usually meaning is extracted from the correlation between multiple events from the stream and also from other sources.
* **streams** may be homogeneous, i.e. contain a single type of events, or heterogeneous, i.e. contain multiple types of events.

In BPM, facts from one entry point (stream) may join with facts from any other entry point or event with facts from the working memory. Although, they never mix, i.e., they never lose the reference to the entry point through which they entered the engine. This is important because one may have the same type of facts coming into the engine through several entry points, but one fact that is inserted into the engine through entry point A will never match a pattern from an entry point B, for example.

* + 1. **Declaring and Using Entry Points**

Entry points are declared implicitly in BPM by directly making use of them in rules. i.e. referencing an entry point in a rule will make the engine, at compile time, to identify and create the proper internal structures to support that entry point.

BPM Fusion is the BPM module that is a part of the Business Logic Integration Platform. It is the BPM event processing engine covering both CEP and ESP. Each IRS BPM Decision Systems event has a type, a time of occurrence, and duration. Events also contain other data like any other facts - properties with a name and type. All events are facts but not all facts are events. Events have clear life cycle windows and may be transparently garbage collected after the life cycle window expires, we will be interested only in transactions that happened in the last 5 minutes. Rules can deal with time relationships between events. The interval and expression timers have 3 optional parameters named "start", "end" and "repeat-limit".

The Rules engine used by IRS BPM Decision Systems is an optimistic failure model. That is; a failure is not assumed and not raised unless reported. A lack of conditions (errors); assume a good status.

**How to discard duplicated facts on insertion**

In some scenarios, you will have to discard equal objects (objects of the same type and values) when they are inserted into the working memory, to avoid data inconsistency and unnecessary activations. In this recipe, we will see how to change the insert behavior just by changing one configuration property.

**Service Model Overview**

* Represented as a directed graph in-memory
* Each node in the graph will have a universally unique ID and a “node type”