

Applying Intervals to Actions in a Document Rendezvous Model
to Support Billable Event Rating:

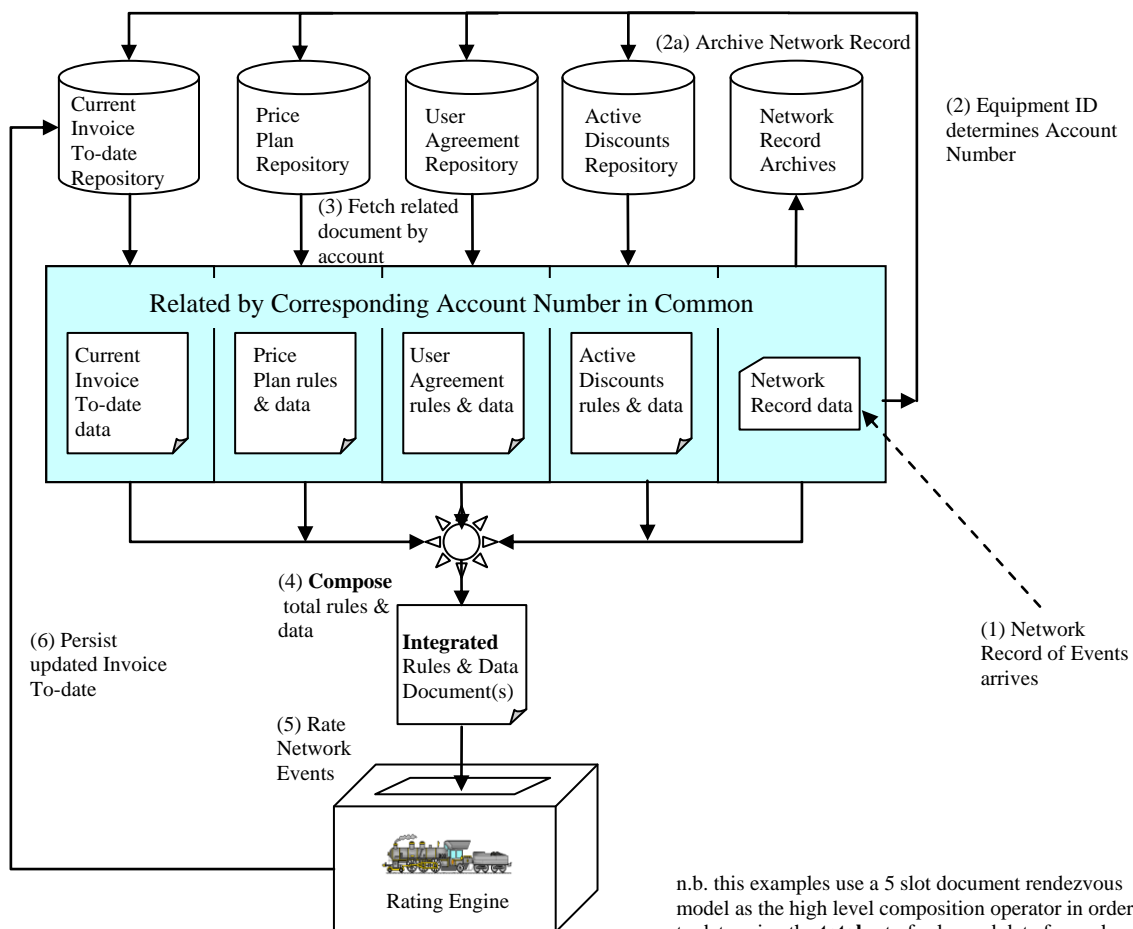
Beginnings of a Compositional Architecture

An Early Example of Compositional Architecture: A Compositional Rule Based Rating Engine

“Law of Compositional Architecture: You can’t easily compose what you can’t easily decompose”

As the following notes elaborate, the business rules for this Telecom Billing and Rating Engine architecture yielded to a procedural scripting language which came to be called “Temporal AWK” for reasons that will shortly become obvious to those familiar with the original AWK language. This procedural attribute was essential for being placed into the high volume, high performance billing pipeline consisting of vast transactions flows. No inference engine was needed, desired, nor could be tolerated.

The question then became how to apply the right rules at the right time on the right data of which there were terrabytes. The scripting language with its notion of temporal intervals literally handled the problem of the right rules at the right time. But the other key observation was that despite the TOTAL database being incredibly huge, the amount of data per transaction was rather small: basically the price plan, the user agreement, the invoice to date, and the incoming network record, all of which could be treated as documents.



n.b. this examples use a 5 slot document rendezvous model as the high level composition operator in order to determine the **total** set of rules and data for each rating transaction

The appropriate documents could be found by following the equipment identification from the incoming network record to the owing account and from there on the other directly relevant documents. These documents formed such a basic unit of information that they could have been kept together and reached by a simple hashing function applied to the account – thus in principle they could be spread across a server farm with the limit of inherent parallelism on the number of useful processors (or threads) being the number of active accounts which numbered in the millions!

In fact most transactional systems are like this: something in each transaction is the key to gathering the other related units of information which represent very small pieces out of a very large ocean of mainly irrelevant (unrelated) data. (I suspect in practice most real transactional systems bind on the performance of monolithic persistent data stores, aka “database”, much more than on the inherent processing load of the transaction flows.)

The compositional insight came from realizing that the transactional documents could be flowed from the originating subsystems (price plan creation, user agreement, current discounts, etc.) to the billing engine as needed with ability of companion documents to reference each other (XPath etc.). This supported the notion of business rule flow (the scripting language) together with data flow between components of the total system, of which the billing and rating engines was only a part. In fact the need for rule flow between major subsystems was spotted as a key requirement before the compositional insight and brought attention to the need to find some manner to enable rule flow.

Rule flow is difficult to nearly impossible for Object Oriented or Component based architectures because they engage in “information hiding” which makes the composition of rules from multiple sources somewhere between hard to impossible. Thus, they tend to be capable of producing high volume, high performance systems - but ones that are dreadfully brittle by being very hard to change. Furthermore, OO and Component architectures typically tend to spread rules rather arbitrarily thought their elements and thus related rules that need to be combined in new ways are hard to even find. When this does happen, it often leads to large scale and painful “refactorings” – something Compositional Architectures easily do by nature.

The real challenge for modern system design is to retain the high volume, performance attributes but also to create highly flexible systems as well. That in a nutshell motivated this early example of using Compositional Architecture techniques: the scripting language contained in documents made embedded rule and associated data flow possible, while the compositional operators were the document rendezvous model based upon the main transactional key and interconnection by means of URI/Xpath references between the selected documents.

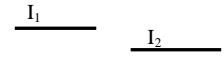
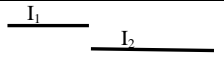
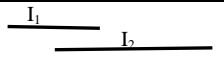
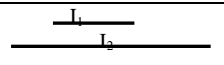
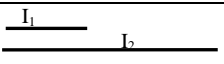
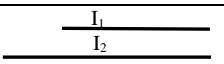
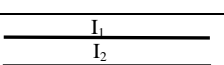
In terms of the ultimate goal of all IT systems, namely the automation of knowledge content, Compositional Architecture emphasizes taking pieces of knowledge wherever they may be located around the network and putting them together into new entities that exist for the duration of their need after which they evaporate. This means the knowledge content of the Network of Interest (NoI), son of the System of Interest (SoI), must be capable of being divided and recombined (composed) as needed.

Barriers that prevent this are simply evil: knowledge entangled in extraneous hideous representations such as 3GLs, OO or otherwise, obscure binary representations, and of course, entrapped by “information hiding”. Although objects can be “composed” in their own limited way by dynamic class loaders, the central testament of object orientation, namely encapsulation, fundamentally limits the utility of the OO approach relative to the Compositional Architecture. While they may cohabit a given design, Object Orientation and Compositional Architecture are fundamentally incompatible approaches. In its own way, this echoes the fundamental “impedance mismatch” between the Relational Data Model and OO.

There is also an implicit recognition that multiple knowledge representations will be present such as XML, RDMS, XLST, general purpose languages such as Java and PERL, and application specific languages. Further more it is generally true that the higher, more abstract, and closer to the problem domain, the better the knowledge representations are for compositional purposes.

Background: Allen Temporal Interval Algebra

- There are 13 fundamental relations that hold between two basic intervals (from “Actions and Events in Interval Logic”, James F. Allen and George Ferguson, 1994).
- In this example logical “intervals” are mapped to real number valued “durations” (other mappings possible).

Basic relation	Inverse Relation	Meaning	Basic ordering
I_1 before I_2	I_2 after I_1		I_1 before I_2 : $I_{1.end} < I_{2.start}$ I_2 after I_1 : $I_{2.start} > I_{1.end}$
I_1 meets I_2	I_2 met I_1		I_1 meets I_2 : $I_{1.end} = I_{2.start}$ I_2 met I_1 : $I_{2.start} = I_{1.end}$
I_1 overlaps I_2	I_2 overlapped I_1		I_1 overlaps I_2 : $I_{1.end} > I_{2.start}$ & $I_{1.start} < I_{2.start}$ I_2 overlapped I_1 : $I_{2.start} < I_{1.end}$ & $I_{2.end} > I_{1.end}$
I_1 during I_2	I_2 contains I_1		I_1 during I_2 : $I_{1.start} > I_{2.start}$ & $I_{1.end} < I_{2.end}$ I_2 contains I_1 : $I_{2.start} < I_{1.start}$ & $I_{2.end} > I_{1.end}$
I_1 starts I_2	I_2 started I_1		I_1 starts I_2 : $I_{1.start} = I_{2.start}$ & $I_{1.end} < I_{2.end}$ I_2 started I_1 : $I_{2.start} = I_{1.start}$ & $I_{2.end} > I_{1.end}$
I_1 finishes I_2	I_2 finished I_1		I_1 during I_2 : $I_{1.start} > I_{2.start}$ & $I_{1.end} = I_{2.end}$ I_2 contains I_1 : $I_{2.start} < I_{1.start}$ & $I_{2.end} = I_{1.end}$
I_1 equals I_2	I_2 equals I_1		I_1 equals I_2 : $I_{1.start} = I_{2.start}$ & $I_{1.end} = I_{2.end}$ I_2 equals I_1 : $I_{2.start} = I_{1.start}$ & $I_{2.end} = I_{1.end}$

- These 13 Interval Relations can also be defined as the following 4x4 table as follows of the ordered numeric relationships between the start and end points of each interval.

Note the upper left and lower right quadrants are invariant, coming straight from the internal definition of an interval. Only the upper right and lower left quadrants actually assert relationships between intervals.

Invariant Quadrants	I ₁ .start	I ₁ .end	I ₂ .start	I ₂ .end
I ₁ .start	=	<=		
I ₁ .end	>=	=		
I ₂ .start			=	<=
I ₂ .end			>=	=

Before/After	I ₁ .start	I ₁ .end	I ₂ .start	I ₂ .end
I ₁ .start	=	<=	<	<
I ₁ .end	>=	=	<	<
I ₂ .start	>	>	=	<=
I ₂ .end	>	>	>=	=

Meets/Metby	I ₁ .start	I ₁ .end	I ₂ .start	I ₂ .end
I ₁ .start	=	<=	<	<
I ₁ .end	>=	=	=	<
I ₂ .start	>	=	=	<=
I ₂ .end	>	>	>=	=

Overlaps/Overlapped	I ₁ .start	I ₁ .end	I ₂ .start	I ₂ .end
I ₁ .start	=	<=	<	<
I ₁ .end	>=	=	>	<
I ₂ .start	>	<	=	<=
I ₂ .end	>	>	>=	=

During/Contains	I ₁ .start	I ₁ .end	I ₂ .start	I ₂ .end
I ₁ .start	=	<=	>	<
I ₁ .end	>=	=	>	<
I ₂ .start	<	<	=	<=
I ₂ .end	>	>	>=	=

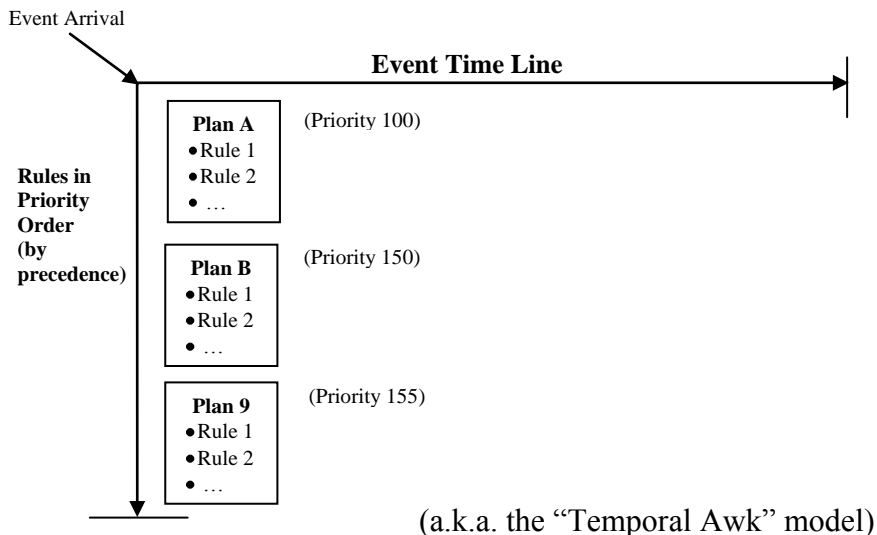
Starts/Started	I ₁ .start	I ₁ .end	I ₂ .start	I ₂ .end
I ₁ .start	=	<=	=	<
I ₁ .end	>=	=	>	<
I ₂ .start	=	<	=	<=
I ₂ .end	>	>	>=	=

Finishes/Finished	I ₁ .start	I ₁ .end	I ₂ .start	I ₂ .end
I ₁ .start	=	<=	>	<
I ₁ .end	>=	=	>	=
I ₂ .start	<	<	=	<=
I ₂ .end	>	=	>=	=

Equals/Equals	I ₁ .start	I ₁ .end	I ₂ .start	I ₂ .end
I ₁ .start	=	<=	=	<=
I ₁ .end	>=	=	>=	=
I ₂ .start	=	<=	=	<=
I ₂ .end	>=	=	>=	=

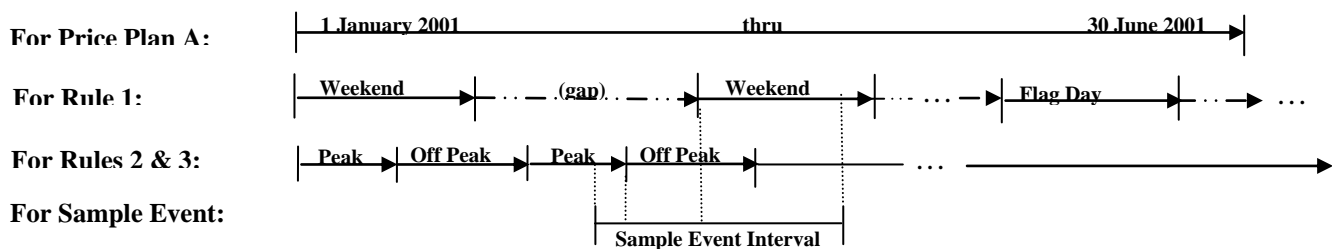
Billing Engine Execution Model

- The use of intervals here does not involve deductive processing or “reasoning”, so “tractability” (huge search spaces) is not an issue
- All rules are strictly procedural, which is essential in a high volume, high performance applications
- Two dimensional control flow: across time and down the rules in precedence order (i.e. non-monotonic rules)
- Time is followed from start of event to end of event
- Rules within a price plan are already in priority from highest (first) to lowest (last) priority
- Price plans are evaluated in order according to explicit priority from highest (lowest number) to lowest (highest number) priority



- Each Price Plan and each Rule is “guarded” by a Qualification Interval which determines the time interval(s) over which the Price Plan or Rule applies, e.g. shown graphically along with the corresponding script fragment and the results of evaluating the Qualification Intervals:

Qualification Intervals:



Price Plan A in effect: <1Jan2001 thru 30Jun2001>
Timeline: CDR_Airtime

<WEEKEND || FLAG DAY>

< PEAK >

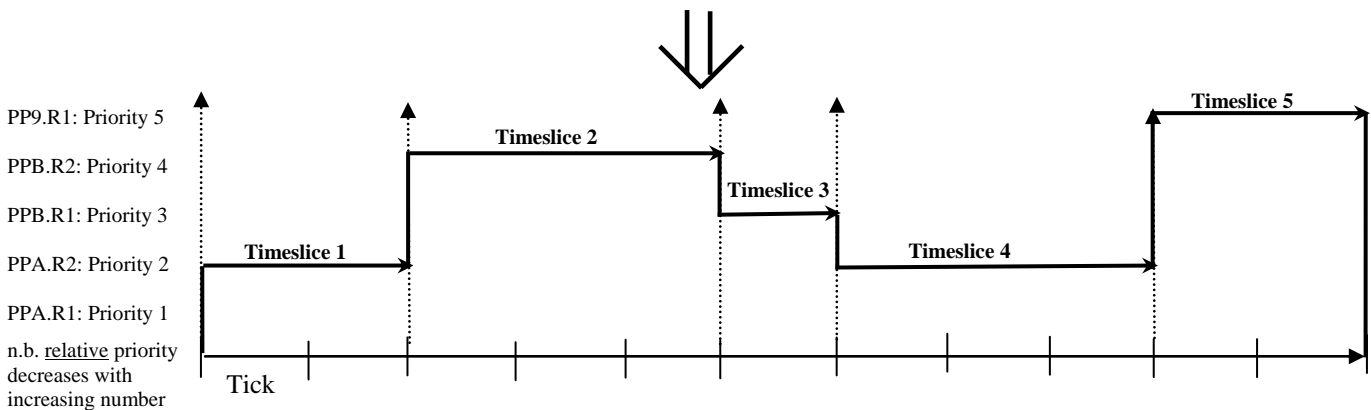
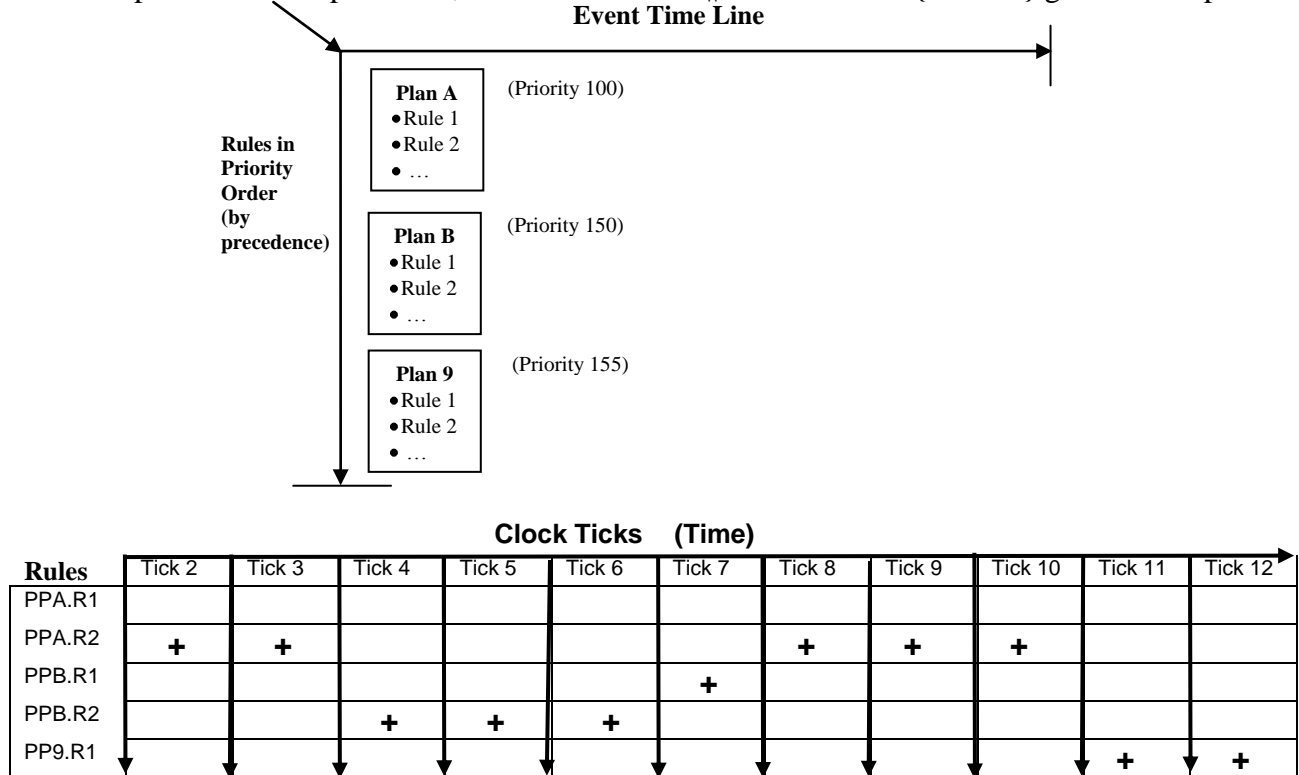
<!PEAK>

{ // Put Rule 1 here for Weekends or Flag Day};

{ // Put Rule 2 here for PEAK time };

{ // Put Rule 3 here for NonPeak time };

At evaluation time <PEAK> { Rule 2} get the first part of Sample Event, <!Peak> { Rule 3} gets the next part of the Sample Event, and <WEEKEND || FLAG DAY> { Rule 1} gets the last part.

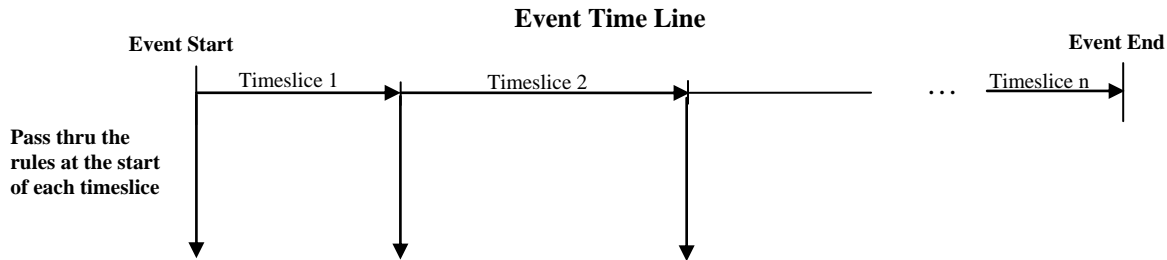


Execution Model: Rule Selection as a (partial) Function of Time

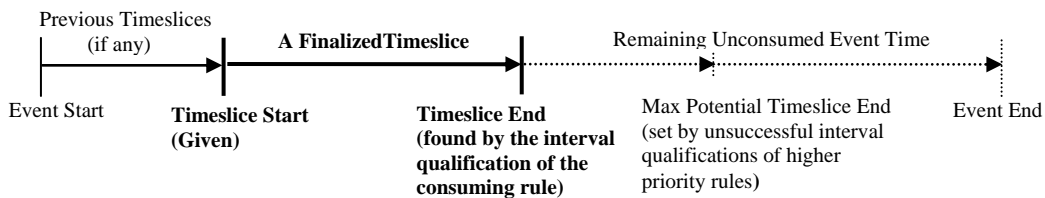
- Rule_{Tick} = Select(Tick, PricePlans, Subscription, Event,);
 - One and only one rule is selected per clock tick
 - Select(Tick, ...) must be a contiguous square wave function
- Combining adjacent ticks which have the same rule leads to naturally to timeslices as event intervals (i.e. the continuous flat parts of the square wave each define a discrete interval).

- This unifies nicely with the notion of Qualification Intervals for Rules
 - Conceptually simplicity
 - Execution can optimally chunk rule application by event timeslice rather than by each individual clock tick (a special case of “compression by exception” optimization)

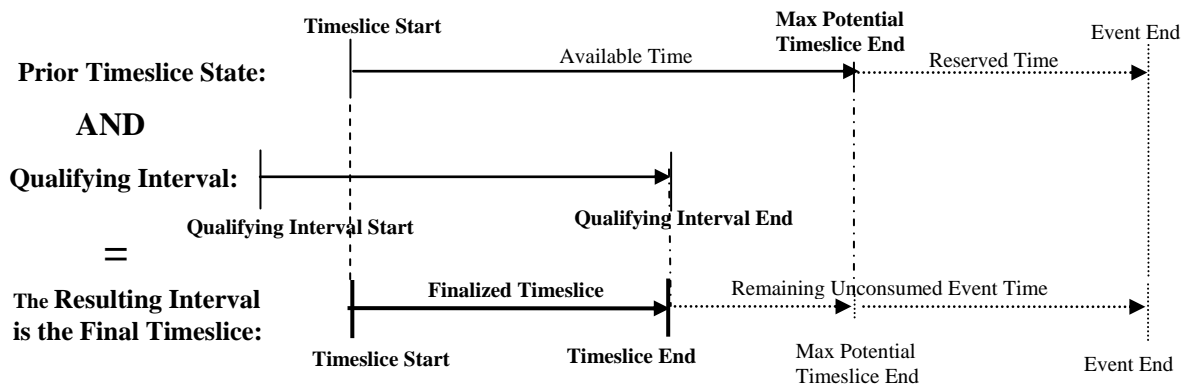
Overall Rate Event Strategy: walk down the event time line by chopping up the timeline into one or more incrementally contiguous rating timeslices until the end of the event time line is reached. Start the first rating timeslice at the beginning of the event time.



Incrementally Contiguous Timeslice Strategy: for each incrementally contiguous timeslice going forward, determine the maximum interval for the current timeslice which matches the first (highest precedence) rule ready to go. Rules are examined once and only once in rule precedence order for each incrementally contiguous timeslice. Initially, assume the current timeslice can go as far as the end of the event.

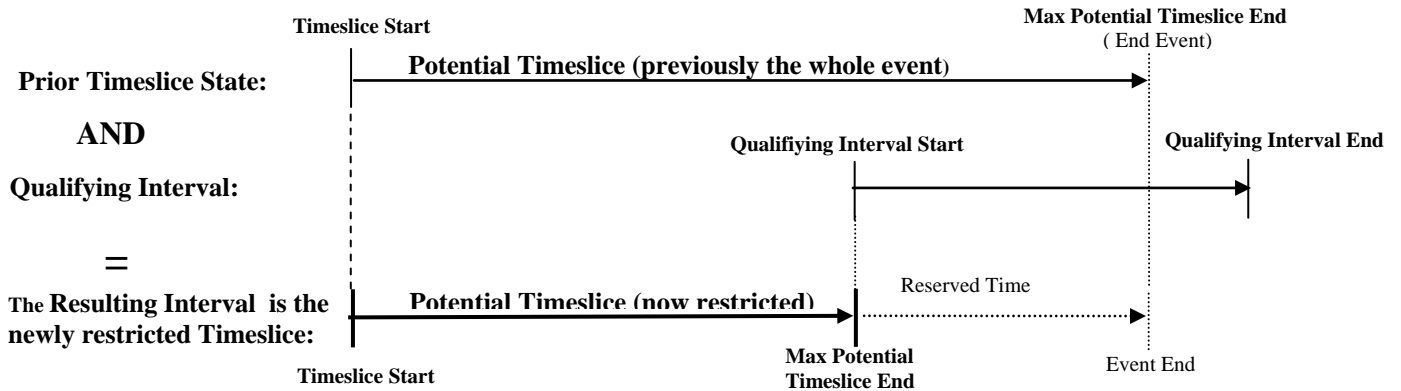


Rule Scanning Strategy: for each rule in order, AND the rule's interval qualification against the current maximum timeslice. The first result which has an interval beginning at the start of the current timeslice consumes the timeslice and also determining its end point. (i.e. the rule with an interval qualification which "overlaps" or "starts/started" the current timeslice).



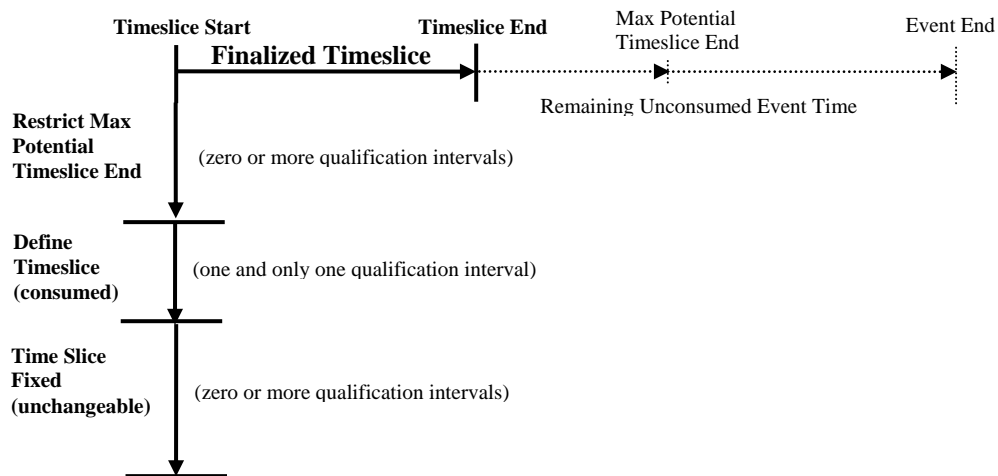
Note: The chosen rule in its action section may shorten the timeslice based upon extra temporal criteria such as \$ threshold exceeded. The rule may not, however, extend the timeslice, except through rounding up.

Until then, resulting intervals which start later than the beginning of the current slice can trim back this timeslice's maximum allowable end point to just before the earliest (smallest) of these starting points encountered. This effectively shortens the maximum allowable duration of the current slice. (i.e. trim the current time slice or that at most it may "meet" the qualification interval which comes "after" the current slice).



Finally, rules encountered after the timeslice has been consumed may view the current time slice, but may not change the bounds of the timeslice.

Graphical Summary of the three major phases of passing thru the rules (Price Plans):



At the end of completely running through the rules, if any time remains in the event, mark the beginning of the next slice to be just past the end of the current timeslice, reset the maximum end of this next timeslice to the end of the event, and do the next iteration of scanning the rules.

And keep going until either the event is fully consumed from start to end, or a gap appears during the scan of a slice indicated by no rule being able to start at the beginning of the slice—which means a time period to which no rule applies: generally, this is a very bad thing and a terribly fatal error.

A Script Language used for Billing and Rating

Notes on Intervals

Basic Interval

- a real number valued segment of the time line with distinct begin and end points
- always the value of start \leq end
- the value of an end point at evaluation time may be expressed by a constant value (absolute time) or may be determined by computing a function
- undef() and/or “Forever” values? And thus duration (end-start) may be defined, or undefined (“finite” or “infinite”)
- the intervals of interest to billing are strictly pro-active (forward ordered: monotonically increasing), never retroactive (running backwards in time)

Bounded Interval

- a basic interval with defined end points, and thus a defined duration

Absolute Interval

- a bounded interval where each end point is either a fixed value or computed by a deterministic function which at most references the other end point, i.e. both values can be computed in advance and mapped to absolute locations on the real number time line

Relative Interval

- a basic interval in which the value of at least one end point depends upon an external parameter, i.e. the function defining the value of that end point references at least one parameter which does not depend upon the value of other end point or a constant value (fixed point in time)

Compound Interval

- one or more basic intervals which meet each other or may contain gaps (which are portions of the real number time line not covered by any of the basic intervals)

Repeating Interval

- a basic interval which happens more than once, i.e. is defined as one or more repetitions of the basic interval
- may contain gaps (can be basic or compounded) the iteration function may be constant or variable

Interval Expressions

op = { “|” | “&” };

“|” (OR)

- At evaluation time, each interval is projected onto the timeline to produce the union of the interval coverages, i.e. coverage is given iff included in at least one interval
- Gaps are permitted, i.e the result is a compound interval (or basic as a special case):
 - for overlapping intervals start: $\min(V_{start})$ and end: $\max(V_{end})$ where “V” means “For All”
 - for non overlapping intervals put start/stop in strict order of the intervals

“&” (AND)

- At evaluation time, each interval is projected onto the timeline to produce the intersection of interval coverages, i.e. coverage is given only iff included in each and every interval
 - for overlapping intervals start: $\max(V_{start})$ and end: $\min(V_{end})$;
 - throw away non overlapping intervals and non overlapping portions of intervals;

“!” (NOT)

- at evaluation time the compliment of the interval is chosen from the timeline, i.e. coverage is given to an interval iff it was a previously a gap, and covered intervals become gaps
- Alternately: think of intervals as bit maps: a basic interval is all “on”, a gap is all “off”. The above operators work as expected on the bit values (i.e as boolean logical expressions).

Sample Price Plan (and notes)

Name: // e.g. NetvisionVOIP
Version: // e.g. 1.3.3 “2001.2 Final Release”
Valid Intervals: // e.g. (30June200 thru 31Dec200) || (1May2001 thru 31Dec2001)
Contexts: // e.g. OnArrival, OnDemand, billCylceFinal, closeOut, etc.
Input Documents: // e.g. NetvisionCDR, Subscriber, Telebuddies
Working Documents: // e.g. standardBillCycleTotals, NetvisionExtras, BezeqSurcharges
Timelines: // e.g. IPBasic, NetToPhoneSrvs

// If more than one timeline, then intervals are referenced by timeLine.intervalName, e.g.
// IPBasic.HOLIDAYS
// NettoPhoneAddOn.PRIME
// IPBasic.{21:00 thru Midnight}
//
// Likewise, for the five built in timeslice descriptors: %SOE,
// %EOS, %Start, %End, %Bound, %Duraton, %Now e.g.
// IPBasic.%EOE
// NetToPhoneSrvs.%End
//
// Similary, for referencing elements in documents which are sensitive to multiple contexts, e.g.
// OnArrival.standardBillCyle.Total\$
// closeOut.NetvisionExtras.YTDMIN
// %CURRENTCONTEXT.standardBillCyleTotals.TotalBillableTime
//
// N.b. Documents must also be defined to match on a given interval(s), which provides
// an automated kind temporal “versioning” for each document type, e.g.
// %ThisBillingPeriod
// withIntheNext24Hours
// MAY2003
// %VictoriaDayLongWeekend
// %AnyTime

ACTION: // e.g. Rate_Event

BEGIN_EVENT { ... };

<Qualification Interval Expression>.logical expression
@) { // QI at least overlaps %Start,
// may optionally move %End backwards
// via onlyuse(duration) or release(duration) };

after) { // QI has first interval later than %Start,
// moves %Bound backwards };

continued) { // Too bad, so sad, the current Timeslice is taken
// may not alter %Start or %End or %Duration };

...

END_EVENT { ... };

Interval built-in variables:

- %BOE - beginning of event
- %Start - beginning of slice
- %Bound - maximum upper bound on slice
- %End - end of slice
- %Duration - length of slice from %NOW to %END
(+ve, 0 (point) or -1 (fully consumed))
- %EOE - end of event
- %DOE - duration of event
- %NOW - current moment
- %DT - default Time Line

Special control statements:

- moveon - end rule processing, skip on to next rule
- nextevt - go to next event cycle, skipping subsequent event
- ignore - ignore this rule match, resetting %End, %Now in @), or %Bound in after), or no-op in continued)
- skip - ignore + next

Interval builtin functions:
(timeline by default
of explicit reference,
eg. tick(Airtime.10))

Special @) functions (initially %Now = %Start):

- tick(length) - %NOW += length (%Now+length <= %End+1)
- useonly() - %End = %Now (%NOW <= %END)

Special after) functions:

- Bound (time) - %Bound = time
(%Start <= time <= previous %Bound <= %EOE)

Example Price Plans

Original Definitions (informal notation)

Product=Basic (aka BasicAirphone)

Priority = 50

Qualifiers=Calendar(Peak=M-F 07:00-19:00,else Offpeak)

Rounding=30sec

PricePlans Actions:

rateEvent: If Peak, Step(0-100,.25,101+,.20)
Else Step(0-50,.15,51+,.10)

Accums=Peak,OffPeak

Product=WeekEndBasic (aka BasicWeekendAirphone)

Priority=60

Qualifiers=Calendar(Sat or Sun)

Rounding=6sec

PricePlan Actions

rateEvent: Tier(0-100,.06,101-200,.05,201+,.04)

Accums=AllWE

Product=Friends&Family (aka Friends&FamilyAirphone)

Priority=65

Qualifiers=

Rounding=30sec

PricePlans Actions

rateEvent: if Friend, then PerMin(.10)

recurringCharge: 9.95

Accums=

Product=30FreeMin (aka Free30Airphone)

Priority=70

Qualifiers=

Rounding=30sec

PricePlans Actions

rteEvent: if FreeMin not = 30, FreeMin = FreeMin+EventMin...

recurringCharge: 9.95

Accums=FreeMin

Product=10FreeWeekendMin (aka WeekendFree10Airphone)

Priority=80

Qualifiers=Calendar(Sat or Sun)

Rounding=6sec

PricePlan Actions:

rateEvent: if WEFree not=10, WEFRee=WEFRee+EventMin...

Accums=WeFree

Product=AllDayMovie (aka AllDayMovie)

Priority=90

Qualifiers=Event=Movie

Rounding=

PricePlan Actions:

RateEvent: if FirstToday(3.99), Else (0.00)

Accums=

+++++

Price Plan BasicAirphone

```
Name:                PP = BasicAirphone;
Version:             1.0    "Initial Examples";
Applies:            1Jan2000 thru 31Dec2000;

Reference Documents: Pdf  = BasicAirphoneProductDefinitionParms,
                   Evt  = stdCDR,
                   Sub  = subscription;

Priority:            Sub.PP.priority, // = 100
                   Pdf.priority;    // = 50

Contexts:           OA  = onArrival;

Working Documents:  SWL = stdWirelessRunningTotals;
                   HST = stdWirelessRatedHistory;

Timelines:          WLT = wirelessTimeline;    // 1st one is always the default timeline

Interval PEAK:      Pdf.peakInterval;    // <07:00 thru 19:00 repeated Monday thru Friday>
Interval OFFPEAK:  !PEAK;                // Local definition

ACTION: rateEvent    // see BasicWeekendAirphone for a better way to handle the step/tier calculation

BEGIN_EVENT        { pt = SWL.peak.basicTimetodate; ot = SWL.offpeak.basicTimetodate;
                   bp = 0;                    // basic billable price ($) for this event (extra info)
                   pi = Pdf.peak.initialMinutes; // 100
                   pr = Pdf.peak.initialRate;   // .25
                   pf = Pdf.peak.finalRate;    // .20
                   oi = Pdf.offpeak.initialMinutes; // 50
                   or = Pdf.offpeak.initialRate; // .15
                   of = Pdf.offpeak.finalRate;  // .10
                   }

<PEAK>.(pt <= pi) // At least some initial Peak Minutes still in effect
  @ )
  { prevpt = pt;
    if ( (pt+=%duration) <= pi ) { first = %duration; second = 0; }
    else { first = pi - prevpt; second = %duration - first; }
    bp += price = (first*pr + second*pf);
    // Update history (HST) entries here : rate type, duration, price, etc. }

<PEAK> @ ) // Billable Peak Minutes only
  { pt += %duration; bp += price = %duration*pf ;
    // Update history (HST.) entries here }

<OFFPEAK> @ ) // Free and Billable Off Peak Minutes together
  { prevot = ot; ot += %duration;
    if ( ot <= oi ) { first = %duration; second = 0; }
    else if ( prevot < oi ) { first = oi - prevot; second = %duration - first; }
    else { first = 0; second = %duration; }
    bp += price += (first*or + second*of);
    // Update history (HST.) entries here }

END_EVENT        { SWL.peak.basicTimetodate = pt;
                   SWL.offpeak.basicTimetodate = ot;
                   SWL.basicPrice = bp; // extended info.
                   SWL.basicPricetodate += bp; // more extended info.
                   // Update history (HST) entries here }
```

Price Plan BasicWeekendAirphone

```
Name:                PP = BasicWeekendAirphone;
```

Version: 1.0 "Initial Examples";
Applies: 1Jan2000 thru 30Dec2001;

Reference Documents: Pdf = BasicWeekendAirphoneProductDefinitionParms,
Evt = stdCDR,
Sub = subscription;

Priority: Sub.PP.priority, // = 200
Pdf.priority; // = 60

Contexts: OA = onArrival;

Working Documents: SWL = stdWeekendWirelessRunningTotals;
HST = stdWeekendWirelessRatedHistory;

Timelines: WLT = wirelessTimeline; // 1st one is always the default timeline

Interval WEEKEND: Pdf.Interval
// Israeli definition = <Friday@thirdStar(Friday) thru [Saturday@thirdStar\(Saturday\)](#)
// repeated Weekly>

ACTION: rateEvent

BEGIN_EVENT { wt = SWL.timeToDate; }

<WEEKEND> @ () { // Generalized solution for tier(0-100,0.06,101-200,0.05,201-*,0.04)
foreach tier in Pdf.tierDefinitions { // walk the tier definitions in the Prod Def. Subtrees
if (tier.upperLimit == undef() | wt <= tier.upperLimit) {
if (tier.upperLimit == undef() | wt+%duration <= tier.upperLimit) { td = %duration; }
else { td = tier.upperlimit - wt; }
price += td * tier.rate;
wt += td;
tick(td);
// Update history (HST) entries here : rate type, duration, price, etc.
} } unless (%NOW > %END)
}

END_EVENT { SWL.timeToDate = wt; }
// Update history (HST) entries here }

Price Plan Friends&FamilyAirphone

Name: PP = Friends&FamilyAirphone;
Version: 4.5 "Post Merger";
Applies: 1Jan2000 thru 31Dec2003;

Reference Documents: Pdf = FFAirphoneProductDefinitionParms,
Evt = stdCDR,
Sub = subscription;
Crc = callingCircle;

Priority: Sub.PP.priority, // = 300
Pdf.priority; // = 65

Contexts: OA = onArrival;

Working Documents: HST = ffWirelessHistory;

Timelines: WLT = wirelessTimeline; // 1st one is always the default timeline

Interval EVERYDAY: Pdf.Interval ; // <Day repeated Daily>

ACTION: rateEvent

EVERYDAY.(isin(Evt.calledID, Crc)) // Rate only if caller is in the subscribers' f&f circle
@)
{ price = Pdf.Rate * duration;
// Update history (HST) entries here }

ACTION: recurringCharge // Parametric data in context ???
@) {price = Pdf.recurringCharge; } //9.95

Price Plan Free30Airphone

Name: pp = Free30Airphone;
Version: 4.5 "Post Merger";
Applies: 1Jan2000 thru 30Sept2002;

Reference Documents: Pdf = F30AirphoneProductDefinitionParms,
Evt = stdCDR,
Sub = subscription;

Priority: Sub.PP.priority, // = 400
Pdf.priority; // = 70

Contexts: OA = onArrival;

Working Documents: SWL = stdF30WirelessRunningTotals;
HST = F30WirelessHistory;

Timelines: WLT = wirelessTimeline; // 1st one is always the default timeline

Interval EVERYDAY: Pdf.Interval ; // <Day repeated Daily>

ACTION: rateEvent

BEGIN_EVENT { ft = SWL.freeMin;
lim = Pdf.FreeTime; }

EVERYDAY.(ft < lim) // Some free minutes left – use up as much as possible
@) { if (ft + %duration <= lim) { ft += %duration; }
else { tick(lim-ft); useonly(); ft = lim; };
// Update history (HST) entries here }

END_EVENT { SWL.freeMin = ft;
// Update history (HST) entries here }

ACTION: recurringCharge // Parametric data in context ???
@) { price = Pdf.recurringCharge; } //9.95

Price Plan WeekendFree10Airphone

Name: pp = WeekendFree10Airphone;
Version: 4.5 "Post Merger";
Applies: 1Jan2000 thru 30Sept2002;

Reference Documents: Pdf = WE10AirphoneProductDefinitionParms,
Evt = stdCDR,
Sub = subscription;

Priority: Sub.PP.priority, // = 500
Pdf.priority; // = 80

Contexts: OA = onArrival;

Working Documents: SWL = stdWE10WirelessRunningTotals;
HST = WE10WirelessHistory;

Timelines: WLT = wirelessTimeline; // 1st one is always the default timeline

Interval WEEKEND: Pdf.Interval
// Israeli definition = <Friday@thirdStar(Friday) thru [Saturday@thirdStar\(Saturday\)](#)
// repeated Weekly>

ACTION: rateEvent // N.b. Eactly the same as Free30Airphone, except different bindings
// of the product definition and the working totals documents.

BEGIN_EVENT { ft = SWL.freeMin; // Count up timer for fun.
lim = Pdf.FreeTime; }

EVERYDAY.(ft < lim) // Some free minutes left – use up as much as possible
@) { if (ft + %duration <= lim) { ft += %duration; }
else { tick(lim-ft); useonly(); ft = lim; };
// Update history (HST) entries here }

END_EVENT { SWL.freeMin = ft;
// Update history (HST) entries here }

ACTION: recurringCharge // Parametric data in context ???
@) { price = Pdf.recurringCharge; } //5.00

Price Plan AllDayMovie

Name: PP = AllDayMovie;
Version: 1.3 "IP Beta";
Applies: 1Jan2002 thru 31Dec2003;

Reference Documents: Pdf = AllDayMovieProductDefinitionParms,
Evt = stdIPDR,
Sub = subscription;

Priority: Sub.PP.priority, // = 600
Pdf.priority; // = 90

Contexts: OA = onArrival;

Working Documents: SWL = stdAllDayMovieTicket;
HST = allDayMovieHistory;

Timelines: IPT = IPTimeline; // 1st one is always the default timeline

Interval EVERYDAY: Pdf.Interval; // <Day repeated Daily>

ACTION: rateEvent

EVERYDAY.(SWL.counter ++ = 0) // All viewings count, but only the first one costs
@) { price = SWL.price;
// Update history (HST) entries here }