**Creating a New Logic Check**

*Updated: 2013-06-04 (Trevor Robbie)*

**Overview**

One of the goals of this specific implementation was to add new checks in a \*quick\* way by the programmer, instead of having to hardcode them and send out application patches each time. Since it is still seen as an extension of programming, the fields can get technical, some of which require knowledge of Java class and field names.

New logic checks are added by inserting the new logic’s specifications (definition) into the appropriate database tables. The first layer of logic abstraction is primaryLogic field, which is used to interpret the rest of the parameters. The parameters can be generally summarized as two “conditions”, which provide the second layer of logic (yet sometimes only one condition is involved). Example: primary logic “COND1\_AND\_COND2” says **there is an issue if** condition1 and condition2 is met. There is a set of fields that describe condition1 and similarly for condition2. This includes a field for an abstract “operator” type. After using the operator, a “negate” field is checked to see whether the operator is negated to ultimately determine the value of the “condition”. For example, a COND1\_AND\_COND2 primary logic combined with an IN\_SET operator and negate=0 for condition1 and IN\_SET operator and negate=1 for condition 2 means: there’s a logic check issue if field1 is in field1values AND field2 is NOT in field2values.

The base Lava code will support a number of standard primary logics and condition operators. If the desired standard logic is not represented yet, it can either be programmed as new standard logic or be completely custom-defined in the Java code, non-standardized.

Logic checks are applied to particular entities. The accepted primary entity types are: instrument, visit, enrollmentstatus, and patient (note: only instrument logic checks are supported as of now). Each of these is connected to the entity by using the CrmsLogicCheckIssue table. Furthermore, conditions (1 and/or 2) may require identifying an entity’s classname beyond its primary entity type (e.g. primary entity type = instrument, classname = UdsSubjectDemo). Classnames must be subclasses of the primary entity type.

Unless otherwise noted, if an operator deals with any metadata (negative values) or null values, it is NOT considered a logic check issue (regardless of “negate” logic), and is ignored. When programming new logic, if metadata isn’t to be ignored, be sure the “negate” logic has consistent meaning.

Note that (especially with instrument logic checks) the same logic check may have multiple entries into the tables. This is because one parameter may be different depending on how entities are filtered. For example, for instruments, the itemNum parameter may be different for different visit types or instrument versions, thus a separate entry needs to be added for each itemNum possibility (this specific redundancy can resolved in the future by having an applicable data dictionary for lookups).

**Steps to create a new logic check definition:**

1. INSERT a new entry into the `logiccheck` table. Fill in the values appropriately.
   1. CheckDefId – the primary key.
   2. codeId – this must be unique for all checks of the same entity1 classname (e.g. different for all UdsSubjectDemo forms). This will be displayed to the user as the code id unless specified elsewhere, which it will for instruments (see instrumentlogiccheck.checkCodes).
   3. enabled – if logic check is to be applied at all, or completely ignored. Disabled checks could be useful in case it eventually may be applied one day. If previously enabled, a disabled check will remove all issues associated with it during activation time.
   4. isalert – 1 if the check is an alert; else an error. Alerts can be designated as “verified” so that the logic check issue is non-problematic. Errors are always problematic.
   5. checkDesc – description seen by the user to figure out the details of the check. Leave as NULL to get the default description, based on primaryLogic, cond1operator, and cond2operator. Assign to override the default with a special description.
   6. primaryLogic – a string designating the first layer of abstract logic to be used. Depending on the primaryLogic, the fields representing condition 1 and condition 2 are interpreted appropriately. If need new logic code, a developer needs to create it and add it to the list of primaryLogic below.
      1. CUSTOM = doesn’t fit any other primaryLogic because logic is too complex, or the logiccheck table cannot define enough parameters for the logic (limit of two). Instead, the Lava code uses the checkCode to uniquely identify the check.
      2. COND1 = an issue if condition 1 is met
      3. COND1\_AND\_COND2 = an issue if condition 1 and condition 2 is met
      4. COND1\_EQUALS\_COND2 = an issue if condition 1’s result is the same as condition 2’s result
      5. COND1\_NOTEQUALS\_COND2 = an issue if condition 1’s result is NOT the same as condition 2’s result
   7. cond1operator – a string describing condition 1’s main logic. Depending on the condition operator, the rest of the condition 1’s fields are interpreted appropriately. If need new logic code, a developer needs to create it and add it to the list of operator types below.
      1. The following operators can be used for any entities:
         1. IN\_SET = operator returns true if field value is within fieldvalues. This can be used with special fieldvalues expressions to represent many possibilities, such as “less than 40” (field1values=”0-39”) [see field1values below].
         2. ISBLANK = operator returns true if field value is blank, NULL, or has metadata.
         3. ISKNOWN = operator returns true if field value is considered “known”. Metadata and NULL values are considered unknown, along with any value in fieldvalues.
         4. EQUALS\_STRING\_LENGTH = operator returns true if field value is of string length equal to “fieldvalues” (interpreted as a number)
         5. COUNTIF\_IN\_EQUALS\_COUNT = operator returns true if the number of fields in “fieldname” (a list of field names separated by commas) whose values are in fieldvalues comes out to be in fieldvalues\_alt. The condnegate field applies to the count itself, so if negated, the operator returns whether the \*count\* is not in the fieldvalues\_alt.
      2. Instrument-only condition operators:
         1. IN\_SET\_PRIORVISIT = operator returns true if any prior visit’s field value (of same entity1classname) is within fieldvalues. **This cannot be used with any condition 1 logic**, since condition 1 must refer to the primary entity.
         2. KNOWN\_PRIORVISIT = operator returns true if any prior visit’s field value is considered “known”. Metadata and NULL values are considered unknown, along with any value in fieldvalues. **This cannot be used with any condition 1 logic**, since condition 1 must refer to the primary entity.
         3. TOO\_DIFFERENT\_ANY\_PRIORVISIT (or different at all) = operator returns true if \*any\* prior visit’s field value is more different than the value in the primary entity by an amount of “fieldvalues” (interpreted as a number), i.e. if absolute difference is greater than “fieldvalues”. Note: this can also be used to check if any prior visit’s field is different at all (fieldvalues=0). The “fieldvalues\_alt” is used to designate “unknown” values, which are to be ignored (not real). If field value is not real (e.g. NULL or metadata) or no prior visits have real data, then never an issue. **This cannot be used in condition 2 yet.**
         4. TOO\_DIFFERENT\_LAST\_PRIORVISIT (or different at all) = same as TOO\_DIFFERENT\_ANY\_PRIORVISIT, but only checks last visit. **This cannot be used in condition 2 yet.**
         5. EQUALS\_FIRSTVALUE\_PRIORVISIT = operator returns true if the field value is equal to the field value of the first visit containing a real value (non-NULL and not metadata). If current visit’s value is not real, yet there is prior real data, then return false (if condition is negated, then thus becomes an issue). If there is no real prior data, then never an issue. **This cannot be used in condition 2 yet.**
         6. LARGEST\_PRIORVISIT = operator returns true if the field value is the largest (or same as largest) value of all supplied values up to this point. The fieldvalues can be used to designate “unknown” values, which are to be ignored (not real). If field value itself is not real (e.g. NULL or metadata) or no prior visits have real data, then never an issue, regardless of negate logic. **This cannot be used in condition 2 yet.**
   8. cond1negate – 1 if we are to negate the result of the operator in determining the condition result. Else, the condition result equals the operator result.
   9. entity1classname – the Java simple class name designating the data model of the entity of this check. Only entities matching this class name apply this logic check. For condition 1 ONLY, this also represents the class name of the \*primary\* entity to apply the logic. Condition 2 is allowed to specify a different class name for its field. See programmer if class name is not known.
   10. field1name – the exact \*programmatic\* field name of the field; capitalization is important. This can be found in the entity’s Java class property. This is used to find the field’s value.
   11. field1itemNum – the \*user-friendly\* reference to the field, so that users can reference the field in the form. An item number is typically the number of the field on the form itself. Also, for field1itemNum only (not field2itemNum), it immediately precedes the description output, and is used to sort lists. This could even display a range of item numbers, and does not need field1name to be specified.
   12. field1values – the string value assigned here is dependent on the condition operator. If numerical, special characters can be used, applied in following order:
       1. Special variables that can be used (or avoided if not intended). If the character string is found (minus the quotes below), it will be replaced with its value during the check.  
          Special variables:
          1. “age” = represents age at visit (only applied to instrument logic checks)
       2. To designate a **basic** evaluation, place inside parentheses. Only one plus or one minus can be used, and no nested parentheses. This is commonly used with special variables.
       3. To specify multiple values, use commas.
       4. To designate a range of values, use a dash (‘-‘).
   13. field1values\_alt – an alternate, secondary field1values to be used in few of the operators.
   14. cond2operator – same as cond1operator, but for condition 2.
   15. cond2negate – same as cond1negate, but for condition 2.
   16. entity2classname – same as entity1classname, but for condition 2. If different than entity1classname, then will define a cross-form check. If same, then this is optional and can be left as NULL.
   17. field2name – same as field1name, but for cond2 or secondary field for cond1 (for a select few cond1 operators).
   18. field2itemNum – same as field1itemNum, but for cond2 or secondary field for cond1.
   19. field2values – same as field1values, but for cond2 or secondary field for cond1.
   20. field2values\_alt – same as field1values, but for cond2 or secondary field for cond1.
   21. activeDate – the date that a logic check becomes actively used by Lava. If NULL, then Lava ignores this logic check. If non-NULL, then Lava uses it. When creating a new logic check, assign this as NULL. “Activation” is done later in the app (see the later step below), which will apply all inactive checks to applicable entities and assigns the active date.
   22. notes – any notes associated with this check; only seen by those accessing it on the back-end
   23. scope – scope of logic check, like viewproperty and list tables. This does not provide logic check overriding though. It is intended to separate each ADC’s logic checks. E.g. for shared UDS logic checks, put “crms-nacc”.
   24. modified – modified date. Leave empty while creating new check. Ends up being the INSERT date since logic checks are only changed on the back-end, not through Lava.
2. Depending on the entity that is performing the logic check, you may have to INSERT an entry into another table.
   1. If entity is an Instrument, INSERT into `instrumentlogiccheck`.
      1. CheckDefId – the CheckDefId of the same check in the `logiccheck` table. It is a foreign key.
      2. instrVers – comma-delimited list of instrument versions that this check gets applied to. Instruments having any other version would not apply this check. Each version would match `instrumenttracking.instrVer` exactly.
      3. visitTypes – comma-delimited list of “visit types” that this check gets applied to. Instruments belonging to other visit types would not apply this check. For convenience, a match is found if the `visit.visitType` **starts with** any of the strings listed in this field. Be sure this is unique enough for the instrument this is for (based on entity1classname). Note: if it is a PRIORVISIT check, then initial visits should not be included.
      4. checkCodes – a comma-delimited list of user-friendly code strings (e.g. NACC error codes for the same error check). One of them gets displayed to the user to replace the default of logiccheck.codeID value (which is used if this is NULL). Prefix NACC errors with an ‘N’. Prefix Emory errors with an ‘X’. A code of ‘NA’ means “Not applicable” (should never be seen).  
         There will be a code for each combination of instrVers and visitTypes. First list codes of all the visitTypes of the first version, then next version, etc. Example: for instrVers=1,2 and visitTypes=I,F,T, the following would be the ordering of checkCodes: 1I, 1F, 1T, 2I, 2F, 2T. If \*only\* one is listed, this will be used regardless of the number of combinations.
3. Activate the logic check. This is done through Lava. In demo app, is is at Admin action bar: “Activate Logic Checks”. Until then, any inactive logic check definitions are ignored.
4. Ensure there are no obvious errors with the new logic check definition. Search `logiccheckissue` for any issues that have invalidDef set to 1 (SELECT \* FROM logiccheckissue WHERE invalidDef=1). This means the logic check had an error (threw an Exception) while trying to apply it for a specific entity. Check the cond1 and cond2 parameters, and if ok, inform a developer to look into it.