The Automed Constraint Language
Automed Working Document, Version 1

Alexandra Poulovassilis
School of Computer Science and Information Systems, Birkbeck

February 21, 2003

1 Introduction

AutoMed is a data transformation and integration system which supports both virtual and materialized integration of schemas expressed in a variety of modelling languages. See `http://www.doc.ic.ac.uk/AutoMed` for a list of papers and technical reports describing various aspects of the system, and for the current publicly available software.

AutoMed supports a hypergraph-based data model, **HDM**, as the **common data model**. A **schema** in the HDM is a triple $(Nodes, Edges, Constraints)$. A **query** q over a schema is an expression whose variables are members of $Nodes \cup Edges$. AutoMed has a functional query language as its intermediate query language, **IQL** [2]. $Nodes$ and $Edges$ define a labelled, directed, nested hypergraph. It is nested in the sense that edges can link any number of both nodes and other edges. $Constraints$ is a set of boolean-valued queries over the schema which are satisfied by all instances of the schema. Nodes and constraints are uniquely identified by their names. Edges have an optional name associated with them, and are uniquely identified by their name (if any) and the list of nodes/edges that they link.

Higher-level data models and primitive schema transformations for them are defined in terms of the HDM. Schemas are incrementally transformed by applying to them a sequence of primitive transformations $t_1, \ldots, t_n$. Each primitive transformation t_i makes a 'delta' change to the schema by adding, deleting or renaming just one schema construct, defined in some modelling language. Thus, the intermediate (and indeed the target) schemas may contain constructs of more than one modelling language. The **scheme** of each construct uniquely identifies the construct within a schema.

All primitive transformations have an optional additional argument which specifies a constraint on the current schema extension that must hold if the transformation is to be applied.

Each add or delete transformation step is accompanied by an IQL query which specifies the extent of the new or deleted construct in terms of the rest of the constructs in the schema. As well as add, delete and rename primitive transformations, there are also id primitive transformations which assert the semantic identity of a pair of schema constructs, and also extend and contract primitive transformations which behave in the same way as add and delete, respectively, except that they indicate that their accompanying query may only partially construct the extent of the new/removed schema construct; their query may just be the constant Void, indicating that the extent of the new/removed schema construct cannot be derived even partially.

Each primitive transformations, t, has an **automatically derivable** reverse transformation, $\bar{t}$. In particular, each add or extend transformation is reversed by a delete or contract transformation with the same arguments, and vice versa, while each rename or id transformation is reversed by another rename or id transformation with the two arguments swapped. This holds for the primitive transformations of **any** modelling language defined in AutoMed.

The queries within transformation steps allow **automatic translation** of **data** and **queries** between heterogeneous schemas.

In particular, if a schema S is transformed to a schema S' by a single primitive transformation step, the only cases that need to be considered in order to translate a query Q posed on S to a query Q' posed on S' are if the transformation step is a delete, contract or rename, in which case occurrences in Q of a deleted/contracted construct need to be substituted by the query q specified in the transformation, and of a renamed construct by its former name. For sequences of primitive transformations, these substitutions are successively applied in order to obtain the final translated query Q'. For data translation from S to S', it is the add, extend and rename steps that are significant.

In [1] we describe how the above query translation scheme is used within an AutoMed integration network that integrates a number of local schemas into a global schema, in order to derive a view definition for each global schema construct in terms of the local schema constructs. These derivations can then be substituted into a query posed on the global schema in order to obtain an equivalent query distributed over the local schemas. This GAV approach to global query processing is what we currently support within the AutoMed system. However, it is also possible to derive a view definition for each of the constructs of a local schema in terms of the constructs of the global schema, as in the LAV approach, and we discuss this too in [1].

2 Constraints in AutoMed

As may be seen from the above discussion, constraints appear in two places in an AutoMed integration network: (a) within the definitions of higher-level constructs in terms of the HDM, and (b) as parameters to transformation steps.

These constraints can be specified as IQL queries, using either one of a number of predefined operators (see below), or any other arbitrary IQL query. Such constraints can be reasoned with and processed in two different ways:

(i) The constraints within both schemas and transformation steps can be evaluated in the same way as global queries above.

   Wherever in a transformation network a constraint appears, it can be translated into a query over a set of local schemas and can be evaluated as such.

   This allows verification of constraints against the current set of local data sources, and can be supported directly by the global query processor software developed by Edgar Jasper.

(ii) Constraints can also be reasoned with independently of the particular contents of current local data sources.

   This is where the set of predefined operators is advantageous, in providing a language of more limited expressiveness for automated or semi-automated reasoning. This will in useful in a number of ways:

   – For semantic validation of AutoMed schemas and pathways to check for consistency, inconsistency, subsumption etc. of the constraints within them — so far we do not validate the constraints within schemas and pathways and the onus is on the integrator to do this.
   – In the work on the (semi-)automatic generation of equivalences between schemas of different modelling languages, being pursued by Mike Boyd.
   – In Lucas Zamboulis' planned work on using the AutoMed approach and toolkit for data integration for the semantic web.

The collection type currently supported within IQL is the `bag` type. The predefined set of operators useful for expressing constraints includes:

- `++`, denoting bag union
- `--`, denoting bag difference
- `distinct`, which removes duplicate elements from a bag
- a set of aggregation functions that can be applied to a bag of the appropriate type (`max`, `min`, `count`, `sum`, `avg`, `group`, `gc`)

- `member`, which tests for the membership of an element in a bag
- boolean connectives `and`, `or`, `not`
- `intersect` and `sub`, which can be defined in terms of `--`:

$$\begin{aligned} b_1 \texttt{ intersect } b_2 &= b_1 -- (b_1 --b_2) \\ b_1 \texttt{ sub } b_2 &= (b_1 -- b_2) = [] \end{aligned}$$

- `makeCard`n $(scheme, c_1, \ldots, c_n)$, which expresses a cardinality constraint $c_1, \ldots, c_n$ on an n-ary edge identified by *scheme*.

  Here, each c_i is a pair of numbers, denoting the lower and upper cardinality of the participation of the i^{th} construct within the extent of the edge, with `Any` denoting an arbitrary number. The value of n can be greater than or equal to 2 (though in practice most schema edges will be binary, $n = 2$).

  This is a family of functions and we give below the definitions for $n = 2$ and $n = 3$. Note that any comparison with `Any` is defined to return `False`:

```
makeCard2 (scheme, (l1,u1), (l2,u2)) =
    ([x | (x,y) <- scheme;
          let c = count [y' | (x',y') <- scheme; x' = x]
          in  (c < l1) or (c > u1)]
     = [])
    and
    ([y | (x,y) <- scheme;
          let c = count [x' | (x',y') <- scheme; y' = y]
          in  (c < l2) or (c > u2)]
     = [])

makeCard3 (scheme, (l1,u1), (l2,u2), (l3,u3)) =
    ([x | (x,y,z) <- scheme;
          let c = count [(y',z') | (x',y',z') <- scheme; x' = x]
          in  (c < l1) or (c > u1)]
     = [])
    and
    ([y | (x,y,z) <- scheme;
          let c = count [(x',z') | (x',y',z') <- scheme; y' = y]
          in  (c < l2) or (c > u2)]
     = [])
    and
    ([z | (x,y,z) <- scheme;
          let c = count [(x',y') | (x',y',z') <- scheme; z' = z]
          in  (c < l3) or (c > u3)]
     = [])
```

Note that the above operators can be used to express key constraints e.g. that attributes `A1,A2` of a relational scheme ⟨⟨R, A1, A2, A3⟩⟩ are a key for `R`:

```
let p = [(x,y) | (x,y,z) <- <<R,A1,A2,A3>>]
in  p = (distinct p)
```

and foreign key constraints, e.g. that attributes `B1,B3` of a relational scheme ⟨⟨S, B1, B2, B3, B4⟩⟩ reference attributes `A1,A2` of `R`:

```
[(x,z) | (x,y,z,w) <- <<S,B1,B2,B3,B4>>] sub [(x,y) | (x,y,z) <- <<R,A1,A2,A3>>]
```

Finally, we note that the above set of predefined operators is indicative, not prescriptive. It can be extended, either by adding to the set of IQL built-in operators, or by using IQL's `let` construct within a query for a local function definition.

References

[1] E. Jasper, N. Tong, P. McBrien, and A. Poulovassilis. View generation and optimisation in the AutoMed data integration framework. Technical report, AutoMed Project, 2003.

[2] A. Poulovassilis. The AutoMed Intermediate Query Language. Technical report, AutoMed Project, 2001.