Managing the Evolution of Information Systems with Intensional Views

David Colpaert, Kim Mens & Bernard Lambeau
ICTEAM Institute, Computer Science Engineering pole
Université catholique de Louvain, Louvain-la-Neuve, Belgium
Email: kim.mens@uclouvain.be, bernard.lambeau@uclouvain.be

***Abstract*—Like any software system, information systems suffer from structural inconsistencies that may arise during system evolution. Appropriate tools are needed to encode the structural regularities the system should adhere to, and to check conformance of the system against those regularities upon evolution. Taking inspiration from the intensional views approach to document and verify structural regularities in source code, we developed a similar tool to document and verify structural regularities in large databases. Regularities are expressed by the user at a high level in a graphical user interface, and then translated into relational algebra in order to check the regularities over the data. Discovered inconsistencies are presented back to the user in appropriate high-level data views. As a case study, the developed tool was successfully applied to a safety critical information system deployed at a large Belgian university. It is used by the rescue services to accurately locate users based on the location of their IP phones from which an emergency call was made.**

Keywords*—Intensional views, tools, conformance checking, structural regularities, information systems, relational algebra.

I. Introduction

In previous work [1], [2], [3] we proposed the *Intensional Views* approach to document high-level *structural regularities* in the source code of a software system, and to *check conformance* of the source code against those regularities, in order to facilitate various *software maintenance and evolution* tasks. By documenting explicitly some of the coding conventions and idioms that are typically adhered to by software developers, and by providing an automated mechanism for verifying where and which source code entities do or do not respect these regularities, structural quality of the code can be improved and certain bugs can be discovered or avoided.

Essentially, an *intensional view* is nothing but a set of source-code entities (e.g., classes or methods in an object-oriented program) which are structurally similar (e.g., having a similar name, containing the same entities, or being related to other entities in a similar way). Instead of enumerating all entities that make up a view, they are defined by means of an *intension*, that is, an executable description which yields the set of entities belonging to the view. A logic language embedded in a reflective object-oriented programming language proved to be an ideal choice in which to define intensional views over source code entities in the object-oriented language. On top of this embedded logic language we developed a set of tools to facilitate the definition, conformance checking and visualisation of the intensional views.

The original approach described above focused on source code regularities only. However, it occurred to us that the underlying idea of the approach was generic enough to be applicable to any system containing structured information obeying certain regularities, where system evolution may lead to a decay in the conformance of the system to these regularities. In particular, in addition to software systems (and source code in particular), we believe that information systems (and databases in particular), suffer from similar problems. This article relates on a recent experience where we transposed the idea and tools of intensional views to this new application area.

More specifically, we implemented an information system intended to localise IP telephones at a large Belgian university. In this case study, we observed that the data sources used by the system suffered from problems very similar to those encountered by our earlier research on intensional views. I.e., the data sources contain a lot of structured information that obeys many regularities, but which are often not documented explicitly and for which evolution causes inconsistencies to arise in the data with respect to those regularities.

To address this problem we implemented a new tool, strongly inspired by the original *Intensional View Environment* [3], but now adapted and dedicated to databases. Like the original tool suite, this new tool consists of an *intensional view editor* in which the user can declaratively codify structural regularities to be respected by the databases of the information system. These regularities are expressed in terms of high-level intensionally declared views on the databases, and relations between those views. To check conformance of these regularities, the tool translates the high-level views and relations into more low-level relational algebra expressions that can be verified directly on the databases. After performing this verification, the results are reported back to the user in sufficiently high-level views, allowing him or her to easily discover what entities did and did not respect the regularities.

Section II describes our case study in detail. Sections III and IV then describe how we adapted the original idea of intensional views to apply it to codify and verify structural regularities on the data sources of our case study. Section V takes a step back to discuss some of the advantages, limitations and future improvements of this approach and provides some comparisons to the original intensional view approach. We conclude that the similarities between the intensional views approach applied to databases and to source code are striking, and that the approach can probably be applied to many other kinds of systems suffering from a decay of structured information throughout system evolution.

II. Case Study

The information system of our case study deals with localizing IP phones at a Belgian university. In case of emergency calls, the system needs to be able to inform rescue services about the precise location of an accident. To do this, it relies on three sources of data. First, each time an IP phone is deployed, a technician fills out an Excel document to specify the location of this phone. However, the problem is not only that these phones can be moved, but also that users can disconnect from a phone and then connect to another phone somewhere else, while keeping the same phone number. Phone number locations can thus evolve over time.

A more dynamic source of information about phone locations is thus needed. The system uses two additional data sources. The first is the network locations. Thanks to a mapping of port locations for each switch, we are able to localize phones connected to the network. Another source of locations is through the phone central (called MX1) and SAP. While the phone central specifies which phone number is connected to what phone MAC address, the SAP system contains the phone number and office location for each employee.

With these three data sources (deployment, network and MX1-SAP), we would like to localise phones accurately. Unfortunately, many inconsistencies remain between these sources. The main problem is that phone locations are not always the same according to the different sources. It occurs frequently that, according to some source, a phone is said to be in one building, while according to another source it is located in another building several miles away. And this is not the only problem; a lot of other constrains between the data sources are not satisfied either. For example, some phones existing in one source are simply missing from other sources. We faced thousand and thousand of errors of different types. All these errors were carefully stored in logs, but it soon appeared to be impossible to deal with all these inconsistencies manually.

Yet, it remains crucial to have consistent information about the locations, given the dire consequences that a location error could have. Indeed, sending the emergency services to a wrong place could cause them to lose precious seconds. We cannot afford inaccurate location information such as "Well, the victim may be here, or there, or perhaps there".

We thus need a tool to help us detect inconsistencies in the data. The tool should allow us to define and verify a variety of structural constraints on our data sources. For example, we would like to express the constraint that a location (building + office) must be the same in all data sources for a same phone. One solution could be to express such constraints directly in the SQL query language. However, we wanted our tool to be high-level enough to be used by users which are not necessarily SQL experts. Especially since expressing such constraints often requires rather complex or verbose SQL expressions. Also, the results returned by such SQL expressions may not be easily exploitable. Our goal was to develop a tool where a user can express his constraints in an intuitive and high-level way, using a simple GUI, and which would return its results (i.e., discovered inconsistencies) in a way that can straightforwardly be exploited by end users.

Actually, some tools that satisfy some of the above requirements already exist. Query By Example (QBE) [4] could be a convenient way to express constraints such as the above. In fact, it could express most of the constraints we need to verify. However, it would require users to learn the specific QBE syntax. We would prefer a tool where the user does not have to learn a new language or syntax in order to be able to express, understand or detect violations of constraints.

Alternatively, we might directly use the constraint checking provided by SQL. Expressing constraints in terms of *uniqueness*, *primary key*, *not null* or other SQL constraints could already prevent a lot of inconsistencies. A problem we have, however, is that we need to be able to keep all original data, even though some of it is currently inconsistent. With upstream SQL constraint checking, the DBMS would simply deny such data, causing a loss of essential information that could have allowed us to discover the root causes of the detected problem more easily.

From the above analysis, we concluded that no existing tool seemed to satisfy all of our requirements, but that a tool akin to the original intensional views tool was probably what we needed to solve our problem. Indeed, intensional views for source code allow end users to define, in a high-level way, using an intuitive GIU, structural regularities on source code. These regularities are then translated to logic and verified over the code seen as a logic repository.

Database constraints could also be expressed and verified using first-order logic [5]. Theoretically, constraints could be specified by the user in an appropriate GUI and then automatically translated into propositional formulae, and then further into SQL. In practice, however, relational algebra may provide a more flexible intermediate language than first-order logic expressions. Indeed, relational algebra is closed over relations (every operator takes relations as input and produces a relation as output) which yields a naturally composable way for building complex constraints from user input.

When porting the intensional views approach to the domain of databases, we therefore decided to translate the high-level structural regularities on the databases into relational algebra instead of logic. We use the *Tutorial D* language [6] as relational algebra, and the *Alf* tool [7] as concrete implementation. Alf provides support for compiling SQL code from arbitrary relational expressions.

Using examples from our case study, in the next section we will now describe our instantiation of intensional views for information systems, which relies upon Alf for verifying constraints over the data.

III. Proposed solution

Whereas initially intensional views were intended to check structural source code regularities [1], [2], here we want to apply this concept to check constraints on database tables and their tuples.

Figure 1 shows the *intensional view editor*, i.e. the GUI wherein the user would define his constraints on the data. It illustrates how a user can easily define a desired regularity to be respected by the data of two different tables, and how discovered inconsistencies with respect to that regularity would be reported back to the user. In this (simplified) example, we want to express the regularity that the locations of phones

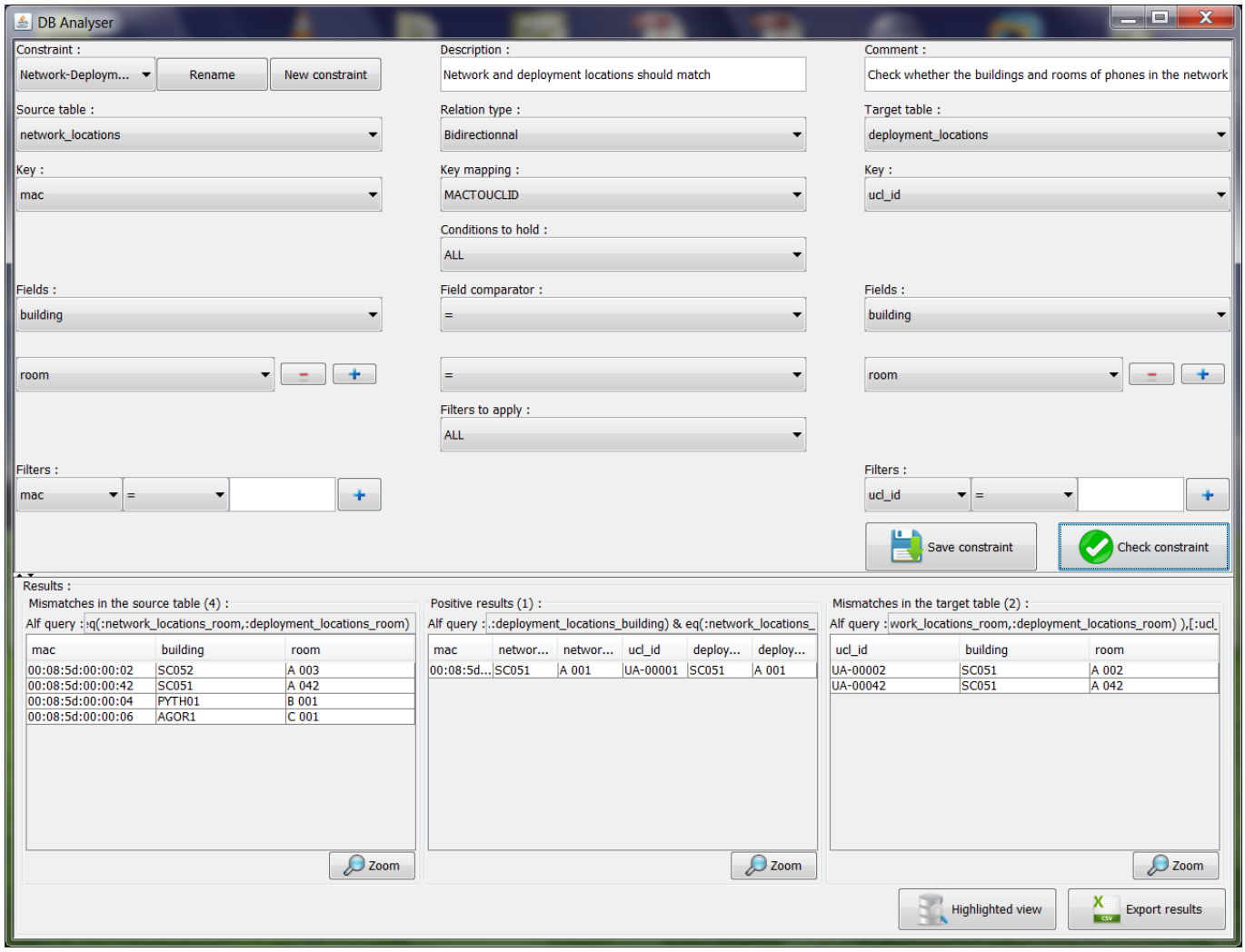

Fig. 1. The *intensional view editor* for defining and checking structural constraints between database tables.

coming from the network data source and from the deployment data source are consistent.

To encode this particular constraint, the user must first select the tables concerned by the constraint. Here, we select the table containing locations from the network and deployment databases. The relation type must also be selected. *Bidirectional* means that all tuples from the source table must have a corresponding tuple in the target table and vice versa. *Left_to_right* means that only each left tuple must have a corresponding right tuple, whereas *Right_to_left* means the opposite.

Next, the way in which the different database tuples should be compared must be defined. The user must select the field needed to identify a corresponding tuple in the source and target table. Typically, this is done by indicating what field in the source and target table represents the identifier or key of that tuple. Sometimes, however, tuples from both tables cannot be directly matched. They can only be matched by using an intermediate table containing a correspondence between identifiers of the source and target table. In our example, because phones in the network table are identified using their MAC addresses, and in the deployment table using their UCL-ID [1], we need to use an intermediate table containing the mapping between MAC addresses and UCL-IDs. This *custom mapping* is encoded in a user-defined predicate MACTOUCLID. Such custom mappings can be defined straightforwardly by a user in a simple XML configuration file. Due to space limitations we refer to [8] for more details on how this is done.

After having selected the concerned tables, their respective keys, and optionally a key mapping, the user can now specify some conditions or constraints on the corresponding tuples between these tables. These conditions can be combined using logical conjunction or disjunction. In order to keep the user interface simple, for now the tool only allows to combine *all* individual conditions with either logical conjunction (ALL) or disjunction (ANY), but does not support a more fine-grained combination or nesting of logical operators. In our example, we define two conditions to encode the constraint that the building AND the room must be equivalent in the two tables. To do this, we require the corresponding fields in the source and target table, i.e. the fields 'building' and 'room', to be equal.

[1]The UCL-ID is a unique identifier given by UCL university to each phone in use at the university.

It is also possible to define some additional filters in order to consider only a subset of the data, for instance, only considering one particular building. This can be useful, for example, when analyzing very large databases with lots of inconsistencies, and the user wants to inspect the inconsistencies for a particular subset of the data only. In our particular example, we didn't apply any such filters.

Finally, when the user clicks on the 'Check constraint' button, three different *Alf* queries are generated. The first one is a query to find the positive results, i.e. all tuples that satisfy the declared constraint. A second query will calculate the mismatches in the source table, i.e. all tuples in the source table that do not satisfy the declared constraint. A third query calculates the mismatches in the target table.

The generated *Alf* query for the positive results looks somewhat like this:

```
restrict(
 join_on(source_table, target_table,
         common_key),
 eq(:source_table_building,
    :target_table_building) &
 eq(:source_table_room,
    :target_table_room))
```

From this query it can be observed that the two concerned tables are first joined based on their common key, and then the results are restricted to the tuples satisfying all conditions, i.e. that the buildings and rooms must be equal. In reality, the actual generated query [2] is a bit more complex than this, to take into account custom mappings (in our example, for instance, there is no common key but the correspondence between MAC addresses and UCL ID's needs to be looked up in an intermediate table), renaming (for instance, when two corresponding fields have a different name in the different tables), and filters (an extra restriction based on the specified filters should be applied).

Each of these generated queries are then executed through *Alf*. As exemplified by Figure 1, positive results are displayed in the table at the bottom center of the GUI, whereas negative results are shown on the bottom left and right, respectively. (For non-bidirectional relations there will no table either on the left or on the right.)

In our example, we see that only one phone (the one with MAC address 00:08:5d:00:00:01 and UCL-ID UA00001) satisfies the constraint of having the same location in both sources. For all other phones, we find inconsistencies and they thus end up in the negative results. A negative result means that either the building or room was different in the other table, or that no correspondence whatsoever was found for this phone in the other table.

Whereas the presented positive and negative results already provide a lot of useful information about detected (in)consistencies in the data, they are not always easy to interpret by the end-user because they are not shown in the context of the original tables. For this purpose, our tool provides an alternative *highlighted view* which simply highlights the detected (in)consistencies in the original tables. To open this view it suffices to click on the button 'Highlighted view' at the bottom of the intensional view editor.

[2]More details on the query generation process can be found in [8].

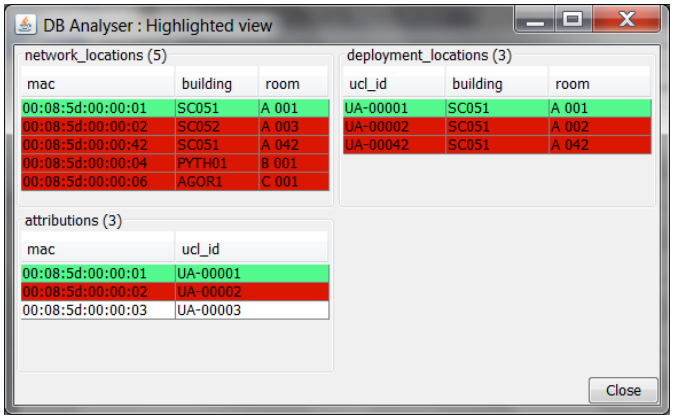

Fig. 2. Inspecting data (in)consistencies with the *highlighted view*.

Figure 2 illustrates what this highlighted view would look like for our previous example. It displays each of the concerned tables, that is, the source and target tables but also the intermediate table used for defining the key mapping. For each of these tables the tuples are coloured either in red if they correspond to an inconsistency, in green if they correspond to a positive result, or just appear in white if the tuple is not concerned by this particular constraint.

In our example, we see that three tables are concerned. The locations from the network and from deployments, but also the intermediate attribution table which maps MAC addresses to phone IDs. The only positive case appears in green, all others in red. One element in the attributions table appears in white because no element in either the network or deployments table had such MAC address or UCL-ID.

Using the highlighted view we can observe, for instance, that the information for the phone with MAC address 00:08:5d:00:00:02 and UCL-ID UA00002 is inconsistent, since it appears with location SC052–A003 in the network table, whereas it has location SC051–A 002 in the deployments table.

IV. Validation

As explained above, intensional views allow the end-user to declare high-level constraints between data sources with relative ease, and reported inconsistencies can then be inspected in two different views to help him identify the causes of the inconsistencies.

In our actual case study, containing the data for about 6500 phones, many inconsistencies were found, such as missing phones, missing information for a given phone, missing mappings between phone IDs and their MAC address, and unknown buildings. All these inconsistencies can be found with our tool. The amount of phones dealt with and the amount of inconsistencies discovered were simply too large to be handled manually, which was the prime the reason for creating this intensional view tool for analyzing data inconsistencies.

For the constraint declared in the previous section, for example, when applied to the 6500 IP phones in use at the university, comparing locations in network and deployments

returned 68% of inconsistent locations (either building or room). Only 13% of the 6500 phones had exactly the same location (building and room) in all sources. Whereas the underlying reasons for all this inconsistencies varied, an automated tool like ours to identify and inspect these errors was a crucial tool to start solving the inconsistencies.

Before putting our tool to use, the approach used was to merge all different data sources into one huge table. But this approach was infeasible due to the many inconsistencies between the data sources. Producing the merged table (which had to be done regularly because of the dynamic nature of some data sources) also took about 5 minutes, whereas keeping the data in their original sources but checking all regularities between them only took a few seconds. Another advantage of defining many different regularities against which to check conformance, was that it makes it easier to isolate and detect certain types of inconsistencies, as opposed to when having to discover inconsistencies in a huge merged database table.

V. Conclusion and future work

Our main contribution is the idea of combining intensional views with relational algebra, to address the problem of managing the structural consistency of an evolving information system. The intuitiveness and simplicity of intensional views match well with the expressive power of relational algebra.

The idea was implemented in an actual tool and put in practice on a non-trivial case study at a large Belgian university, where it is still in use today. The current implementation is only a first prototype, however, and many improvements can still be made. One of its most important limitations is probably that it can only express constraints between two tables, optionally connected through an intermediate table for mapping keys. While this proved to be sufficient for our case study, the tool could be extended so that constraints on multiple tables can be expressed, at the risk of making the GUI less intuitive to use. This is also one of the reasons why the original Intensional View Environment supports intensional relations over two intensional views only.

A similar remark holds for the combination of conditions. The tool currently allows to combine all conditions only with either a conjunction or a disjunction, but doesn't support a finer-grained combination or nesting of logical operators. This too was a deliberate choice because it sufficed for the constraints we needed to express on our case, and because it keeps the user interface simple.

The original Intensional View Environment also offers a notion of *alternative views*, which are useful to define interesting constraints on a single view. We could explore how to use this idea to define unary constraints on a given data source, for example to express that all phones in a given building should have a MAC address with a similar prefix.

We could also improve how custom mappings between tables are expressed. Currently, they are expressed in XML files and only allow mapping the field of one table to the field of another table. Allowing a user to define his own predicates directly in *Alf* would provide more expressiveness. However, this would create a strong dependency of our solution upon *Alf* and contradicts our goal of not requiring the user to know a particular query language. An alternative is to offer this possibility only to expert users, while providing to the average user a dedicated interface for creating custom mappings, which generates the necessary *Alf* query.

An open question remains upon what underlying language our tool should rely. We already motivated our choice for using relational algebra, and *Alf* in particular, but using logic instead (like the original intensional views approach), is possible too, as well as to directly generate SQL queries. Nevertheless, using *Alf* was a good choice from the point of view of ease of implementation and efficiency. It could be worthwhile exploring whether the original intensional view approach couldn't rely on relational algebra as well, instead of upon logic.

Our current approach does not allow to express constraints requiring aggregation, such as "An office cannot contain more than 10 phones". For this, we would need to use aggregation functions such as sum, count or avg. Such functions already exist in *Alf*, and could be added to the tool using some kind of quantifiers. The original intensional view approach did allow for quantification over views, but only universal and existential quantification, with its obvious interpretation in logic.

A further improvement requested by our end-users is to make the GUI more ergonomic. E.g., it could come with a wizard to help novice users define their constraints step by step. We should also add support to make it easy to find out, for a given data element, what constraints it does not satisfy. The original intensional view approach offered such support by integrating the tool in an existing development environment. By analogy we should provide a seamless integration of our current tool in an database management system.

The current paper is a nice case of cross-fertilisation research, where proven ideas of one domain (source code maintenance) are applied to another domain (database maintenance). A similar approach could even be used to any other domain dealing with structured information and suffering from problems due to implicit structural regularities not being respected upon evolution.

References

[1] K. Mens, B. Poll, and S. González, "Using intentional source-code views to aid software maintenance," in *International Conference on Software Maintenance (ICSM 2003)*. IEEE Computer Society, 2003, pp. 169–178.

[2] K. Mens and A. Kellens, "Towards a framework for testing structural source-code regularities," in *International Conference on Software Maintenance (ICSM 2005)*. IEEE Computer Society, 2005, pp. 679–682.

[3] K. Mens, A. Kellens, F. Pluquet, and R. Wuyts, "Co-evolving code and design with intensional views: A case study," *Computer Languages, Systems & Structures*, vol. 32, no. 2–3, pp. 140–156, 2006.

[4] M. M. Zloof, "Qbe/obe: a language for office and business automation," *Computer*, vol. 14, no. 5, pp. 13–22, 1981.

[5] R. Elmasri, *Fundamentals of database systems*. Pearson Education India, 2008.

[6] C. J. Date and H. Darwen, *Foundation for object/relational databases: the third manifesto: a detailed study of the impact of objects and type theory on the relational model of data including a comprehensive proposal for type inheritance*. Addison-wesley, 1998.

[7] B. Lambeau, "Alf, relational algebra at your fingertips," 2013. [Online]. Available: http://www.try-alf.org/ or http://github.com/alf-tool

[8] D. Colpaert, "Un outil de gestion d'incohérences de données basé sur les vues intensionnelles et l'algèbre relationnelle appliqué à un cas de localisation de téléphones IP," Master's thesis, Université catholique de Louvain, 2014.