Towards a formal validation of ETL patterns behavior

Bruno Oliveira[1], Orlando Belo[2], Nuno Macedo[3]

[1] CIICESI, School of Management and Technology, Porto Polytechnic, Portugal
[2] ALGORITMI R&D Centre, University of Minho, Portugal
[3]HASLab, INESC TEC & University of Minho, Portugal

bmo@estgf.ipp.pt, obelo@di.uminho.pt, nfmmacedo@di.uminho.pt

Abstract. The development of ETL systems has been the target of many research efforts to promote their implementation, addressing not only the time and effort required but also means to improve its final quality. In the last few years, we presented a pattern-oriented approach to develop these systems. Basically, a patterns is comprised by a set of abstract components that can be configured to enable its instantiation for a specific case, enabling its mapping to a specific physical format that can be used by commercial tools that support data migration processes implementation. Since ETL systems represent complex and very specific data transformation routines, they are considered error prone processes, even when using high-level components. Several operational requirements need to be configured and system correctness is hard to validate, which can result in several implementation problems. In this paper, a set of formal specifications in Alloy is presented to express the structural constraints and behavior of a slowly changing dimension pattern. Then, a specific Domain Specific Language was built and validated according to the specifications and constraints defined in an Alloy model, ensuring the correctness of the configuration provided, and enabling its translation to physical primitives in a consistent state.
Keywords: Data Warehousing Systems, ETL Systems, ETL Patterns, Component-Reuse, Alloy, ETL Systems Formal Specification, and Domain Specific Language.

1 Introduction

The use of patterns is a recurrent practice in most software development areas, in which systems are frequently designed based on existing components, taking advantage of previous knowledge and experience. Nowadays, the importance of reusable practices and the design techniques that promote them is recognized, which contributes to higher software quality and to reduce the time and money spent in its implementation and maintenance. Additionally, patterns enforce the use of well-proven practices that represent the knowledge of broadly accepted standards or techniques [1], carrying the experience between projects and contributing to its final quality. Currently, software reuse is enhanced through the use of self-contained patterns provided by software libraries, frameworks or service-oriented systems that can take the shape of simple functions to support simple tasks or of configurable components that can be adjusted to specific needs.

The development of ETL (*Extract-Transform-Load*) [2] processes for *Data Warehousing Systems* (DWS) are a specific software area that addresses very specific needs representing a very critical component to any DWS [2]. Each DWS implementation serves its own user community with different ways of acting and thinking, linked to a specific set of business and decision-making processes supported by specific data models. These characteristics render them user specific data clients and even with standard solutions for DWS implementation (which happens often in areas such as telecommunications or retail), several adjustments need to be performed to the original data models to answer to the requisites of those decision makers. Additionally, ETL designers frequently deal with legacy systems that provide limited mechanisms for data extraction, data inconsistencies resulting from years of business change or evolution, and limited time-frames for data processing. All these require extreme care and concern from the ETL architects and software engineers in its planning, architecture, design, and implementation. Additionally, current commercial tools that support ETL/data migration implementation processes

[1] Please note that the LNCS Editorial assumes that all authors have used the western naming convention, with given names preceding surnames. This determines the structure of the names in the running heads and the author index.

provide specialized transformation tasks closely related to SQL operators. This approach contributes to the development of very complex processes comprised by hundreds of tasks, represented using proprietary notations and implemented according to specific architectures and philosophies. These practices are by nature error-prone and hard to maintain, resulting in inconsistent processes. Thus, and despite the effort of several specialists in the area, a simple and reliable approach to simplify this kind of processes is still missing. Since the development of ETL processes shares several development phases and typical problems related to the other types of software, we believe that a pattern oriented approach can be applied through the identification of recurrent procedures or techniques applied in common scenarios, identifying the cluster of operations needed and abstracting its behaviour to provide its instantiation for specific cases. These patterns can be applied to the whole ETL life cycle, supporting users both in initial development phases and in its physical implementation. Basically, an intermediate layer is provided to separate technical knowledge typically used in ETL commercial tools from the domain-knowledge used by decision-makers. To support the complexity of the knowledge involved and the application of each pattern to specific contexts, a first approach to formalize ETL patterns is also presented using Alloy [3], a declarative specification language that supports problem structural modelling and validation, helping to avoid process inconsistencies or architectural contradictions.

Thus, and after a brief summary of related work in Section 2, we demonstrate (Section 3) the feasibility and effectiveness of a pattern oriented approach for ETL development based on the description and formalization of one of the most common used ETL techniques: the *Slowly Changing Dimension* (SCD) with history maintenance (SCD-H). The operational requirements are identified and the respective structural constraints and behaviour formalized using Alloy. Additionally, and based on the structural aspects identified, a *Domain Specific Language* (DSL) pattern instantiation, is shown to provide the necessary bridge to translate a SCD-H pattern to its correspondent execution primitives. Finally, in Section 4, we evaluate the work done so far, pointing out some research guidelines for future work.

2 Related work

ETL modelling has been the subject of intensive research and many approaches to ETL implementation have been proposed, aiming to reduce system design complexity, produce detailed documentation, and provide the ability to easily communicate with business and end users. As far as we know, Köppen [4] firstly presented a pattern-oriented approach to support ETL development, providing a general description for a set of patterns, e.g., aggregator, history and duplicate elimination patterns. This work focuses on important aspects defining patterns for internal composition properties and the relationship between them. The authors distinguish between elementary and composite tasks, presenting composition properties to describing categories of tasks that precedes and proceeds the composite tasks. However, patterns are represented only at design level, lacking the identification of the main configuration components and how each one can be translated to code. With this work, we intend to go further, encapsulating behaviour inside components that can be reused. In fact, we propose a generator-based reuse approach [5] that uses a generator system that can be configured using a specific DSL describing the components of the pattern that should be used and how they work. The generator will produce a specific instance that can represent the complete system or part of it, leaving physical details to further development phases.

In this field, it is also important to mention the work presented by many authors that represent important contributions to the area. Vassiliadis and Simitsis proposed a technique for the conceptual, logical and physical modeling of ETL processes [6], presenting a new set of elements specially designed to express ETL natural features. More recently, Akkaoui [7] presented a work based on MDA (Model-Driven Architecture) for ETL process development, where a vendor-independent model (BPMN4ETL) for unified design and covering the automatic generation of source code for specific computer platforms using a meta-model based on BPMN (Business Process Model and Notation) is proposed. The bridge to execution primitives was explored using a model-to-text approach, supporting its execution through some ETL commercial tools. In subsequent works [8, 9], two important architectural layers were presented for the specification of ETL processes using BPMN: process orchestration, which can be accomplished by several BPMN elements, and data process operations related to some specific operations that allow for the manipulation of data coordinated by control process elements. The author also addressed the problematic of the maintenance phase of ETL life cycle, particularly when business requirements change. They extended the model-driven framework, not only to provide automatic code generation but also to improve ETL maintenance using model-to-text (using the so-called BPMN4ETL meta-model) and model-to-model transformations, respectively, to support both steps. The authors identify structural updates and provide strategies for handling schema changing.

These and other related works [10–12] revealed very important aspects that were taken into consideration for the approach presented here. However, they fail to provide an integrated approach that focus on the complete ETL lifecycle and to take advantage of work preformed in initial development to implementation phases. Additionally, they focus on very granular tasks, resulting in very large and disorganized process that tend to reveal process inconsistencies and redundancy.

3 ETL patterns: SCD-H pattern

Patterns have been used in several software development areas as a way to help developers solve recurring problems, promoting the sharing of experience and knowledge obtained across several areas. Several types of patterns are identified and used by many authors, e.g., the design patterns commonly used to describe a specific problem and a possible solution and should be abstract enough to be reused in many different scenarios; or Generator-based patterns used to reuse algorithms and parameterized using specific domain languages that can be further interpreted and mapped to executable code [1]. Patterns can be viewed as a three-part rule expressing the context, the forces that typically occur, and how the solution [13] resolves the forces [14]. In the next paragraphs, a common ETL procedure is identified and formalized: SCD-H transformation pattern that is used to maintain changes in specific dimensions used to track records history. A common pattern structure, shared by general patterns for software development together with general guidelines for its representation, was used [14, 15]. The general description of the pattern is accompanied by a formalization in Alloy and its specification in a DSL developed to support code generation.

When a data warehouse is updated, some decisions must be considered in order to maintain a consistent view of its data. The SCDs embody well-defined policies that represent design patterns to support old and new data over time in a DW dimension context. Several types of SCD can be applied [2, 16], each one considering specific scenarios. In this work, a specific strategy of SCD that preserves history using a history table is followed. Basically, this approach considers that a dimension is composed by two distinct tables: one stores up-to date dimension's data and the other one stores the previous version of the record values. With this strategy all current and historical records are stored with no limitations and with low process complexity, since they are easier to compute than other approaches [17]. Below, a specific template is used to describe the SCD-H pattern:

Name: Slowly Changing Dimension with history maintenance (SCD-H)
Classification: Transformation pattern
Problem: In some cases, the dimension tables of a data warehouse store data that can change slowly over time. In such cases, there is a need to track these changes to support historical data reporting. How can current and historical data be properly stored?
Context: The DWS are built based on the concept of temporal data. The facts stored in fact tables are linked to a specific date in which they occurred. Not only are fact tables related to the time dimension, but other dimensions can also be affected by the time, i.e., data can only be valid in a certain date interval. This means data from data sources can change to reflect a change in a point of time that should be preserved in a specific dimension, maintaining the consistency of the data warehouse.
Solution: The SCD process begins when changed data is available to populate the target system. Typically, this data is stored using specific audit tables that store records comprised by attributes (*Att*) with history maintenance (*SCDAtt*) for a specific dimension table along with the operation (*Operation*) and the date (*Date*) that was performed, and the table surrogate key (*Skey*):

$$auditData = \langle Skey, SCDAtt_1, \ldots, SCDAtt_n, Operation, Date \rangle$$

Audit tables are used in the *Data Staging Area* (DSA) and provide to the SCD process the records to be processed according to the operation performed in source systems. When records are new (Insert), insert operations are triggered to the dimension table as current data, while the deleted records are marked as inactive in target dimension. A more complex process should be initiated for updated records, since the current versions of updated records should be transferred to history table (partially or totally, depending on the dimension table's SCD specification). Depending on the needs, all or some of these operations can be registered in the system's log file, identifying the record, the operation that was performed, the data source origin and temporal data. The quarantine table also plays an important role for SCD process, since can be used to process unexpected scenarios that don't compromise all process operation. Structural errors or data entry errors can be redirected to quarantine tables that store all inconsistent records, identifying (at least) the record, the error occurred and the temporal data. These records can be posteriorly analysed by specific procedures or even manually in order to be reintroduced in ETL workflow or deleted, helping to identify important scenarios that can be useful to check ETL integrity.

Additionally, log track techniques are used to recover from unexpected situations, since if some critical error happens, the process can restart only from the point when the error occurs, in particular when large volumes of data are involved. After processing all records, the audit table will be cleared and the records processed will be loaded into the target dimensional table.

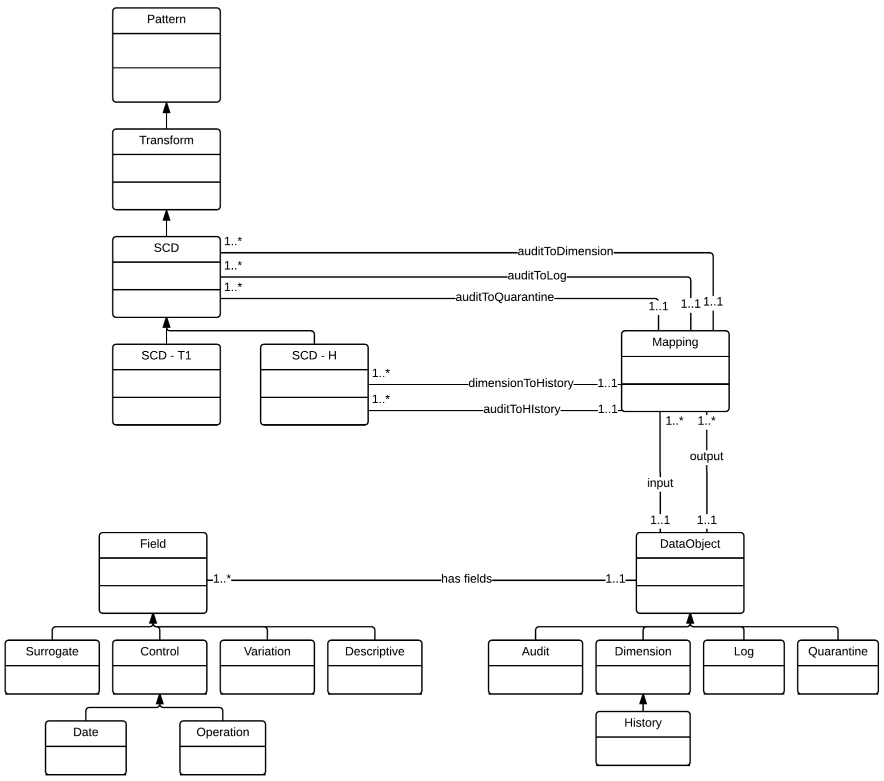

Figure 1. Class diagram for SCD pattern structure representation

To support all these structural and operational constraints a specific model was developed and represented in Figure 1. This model represents a subset of the ETL patterns meta-model for the representation of SCD structure and the related objects that supports its structure. The SCD class is a specific type of transformations patterns that can be specialized in several types. Only two types are distinguished: the type 1 that holds no historical data (data is simply replaced) and history type described before. To support the SCD–H pattern, several metadata should be provided to produce pattern instances. The *DataObject* class represents the several types of data source/target objects that can be used: *Audit* table that holds data extracted from data sources and respective operation and time data, the *Dimension* in which current data will be stored, *History* that stores historical data, *Quarantine* that represents the object that will store non-compliant data and *Log* objects that will store all operations performed. Each one of these objects will have fields from several types, revealing specific operational characteristics of each data object. For example, the dimension object should have *Surrogate Key* (SK) fields and descriptive attributes. Specifically for the SCD-H context, variation attributes should be specified and control fields to represent, for example, the state of each record (to signal active and inactive attributes). Thus, the SCD-H object should be configured using specific mapping that represents the association between data objects. This means that correspondences between fields of each data source should be defined according to the constraints defined for each mapping: the *auditToQuarantine* relationship represents the mapping between audit table and dimension table, which is related to the source of data and target dimension, the *auditToQuarantine* represents the mapping between dimension table and quarantine table to provide storage to the fields that for some reason could not be stored in target dimension and *auditToLog* that represents the relationship between audit table and target log object that will store the operations performed, including the non-successful operations. The *dimensionToHistory* and *auditToHistory* relationships are specified only for SCD specialization class, since represent specific features of SCD

```
abstract sig Field{}
sig SKField, ControlField, VariationField, DescriptiveField extends
Field{}
sig DateField, OperationField, ErrorField extends ControlField{}
abstract sig DataObject{
    fields: some Field,
    keys: some SKField,
    (...)
}
fact dataobject {
    all o : DataObject | o.keys in o.fields
    (...)
}
pred consistentDataObject[s:State,o:DataObject] {
    all r1,r2 : o.rows.s |
        (all f : o.keys | f.(r1.values) = f.(r2.values)) => r1 = r2
    (...)
}
sig Mapping {
    inData: one DataObject,
    outData: one DataObject,
    association: Field -> Field
}
fact mapping {
    all m : Mapping | m.association in m.inData.fields ->
m.outData.fields
    (...)
}
pred consistentMapping[s:State,m:Mapping] {
    consistentDataObject[s,m.inData]
    consistentDataObject[s,m.outData]
    (...)
}
```

Figure 2. A basic Alloy specification for supporting ETL patterns

with history maintenance, representing the mapping between dimension fields to history dimension fields and audit table fields to history dimension fields, respectively.

Although the pattern structure can be described in a straightforward manner in class models like the one from Figure 1, such is not the case for several richer structural constraints that specify the integrity of the pattern, which must be considered not only at the level of the pattern configuration but also by the operational behaviour of the pattern. To provide a simple and solid specification of the SCD pattern amenable to being automatically analysed, its description was embedded into a formal specification. *Alloy* is a lightweight formal specification language, whose *Analyzer* provides support for automatic assertion checking, within a bounded universe, by relying on off-the-shelf constraint solvers. The flexibility and object-oriented flavor of the Alloy language render it well-suited to specify and analyse software design models, addressing both complex structural constraints [18–20] and behavioral constraints imposed by transformations [21, 22]. An embedding of ETL pattern specifications into Alloy would not only provide a formal specification of their structure and behaviour, but would also allow their fully automatic verification, thus ensuring that the pattern preserves the consistency of the system. Figure 2 presents an excerpt of the Alloy specification of ETL patterns and related concepts, formalizing the meta-model structure and providing a new degree of detail to the class model presented in Figure 1.

The class hierarchy can be easily described using Alloy *signatures*, that introduce sets of elements of a certain type in the model. Abstract signatures are used to describe abstract concepts that should be refined by more specific elements, which is the case of top-level signatures *Field*, *DataObject* and *Pattern*. These abstract signatures can then be specialised through extension into concrete objects, mirroring the hierarchy of the model from Figure 1. Signatures may contain *fields* of arbitrary arity, that will embody the associations between the different artefacts of the specification. For instance, similarly to the Figure 1 representation, data objects represent repositories that are used to read and write data and contain a set of field declarations that is shared by its extensions: each *DataObject* element is related to a non-empty set of *Field* elements (imposed by the keyword *some*), that represent data fields used to represent records, and a set of *SKField* elements, used to express the SK fields. These can then be restricted by additional constraints through Alloy *facts*, like the one stating that every *SKField* is selected from the *Field* elements of the data object. Signature *Mapping* represents the association between fields from two data sources, establishing the relationship between attributes from two different sources through the binary field *association*. Additional constraints impose, for instance, that a mapping is only valid if it associated fields from the input (*inData*) and the output (*outData*) data sources. Facts represent constraints that are enforced in the system; however, certain constraints should not be enforced but rather preserved by the

```
sig Audit extends DataObject {
    dateFields: some DateField,
    opsFields: some OperationField,
    varFields: some VariationField
}
sig Dimension extends DataObject {
    dFields: some VariationField
}
fact dataobjects {
    (...)
}
abstract sig SCD extends TransformationPattern{
    auditToDimension : one Mapping,
    auditToQuarantine: one Mapping,
    auditToLog: one Mapping
}
fact scd {
    all scd : SCD {
        scd.auditToDimension.inData in Audit
        scd.auditToDimension.outData in Dimension
        DateField.(scd.auditToDimension.association) in DateField
    (...)
}
sig SCDH extends SCD {
    dimensionToHistory: one Mapping,
    auditToHistory:one Mapping
}
fact scdh {
    all scd : SCD {
        scd.dimensionToHistory.inData in Dimension
    (...)
}
```

Figure 3. Basic Alloy specification to support SCD-H configuration

ETL procedure: these denote the notion of correctness in the specification. The behaviour of the patterns should then be checked to assess whether they guarantee the consistency of the system. These *predicates* include properties like the uniqueness of the values in SK fields and the referential integrity between the SK fields of the data sources related by the mappings.

Figure 2 represent the abstract specification of an ETL pattern, which is expected to be specialised by concrete patterns. The excerpt of such specialisation is presented in Figure 3 for SCD patterns. For instance, data objects are specialised, among others, into *Audit* and *Dimension* signatures, each containing specific fields that express the operational properties related to the audit and dimension sources used in SCD patterns. For audit objects, date fields specify the temporal data for the action performed in the source systems, operation fields the operation performed and variation fields specific version of data. Again, these fields are forced to be selected from the *fields* of the data object, providing a level of validation that class diagrams cannot express. Other relevant signatures, omitted in the figure, are specified in a similar manner, like those regarding the target and historical dimension and their respective constraints, as well as the log and quarantine objects.

The pattern signature is then specialised to represent SCD patterns, containing the expected mappings, like *auditToDimension*, *auditToQuarantine* and *auditToLog*. These fields describe the relationship between the audit object used for the SCD process and the respective target repositories: a dimension object that will receive data from audit table, a quarantine object that holds non-conformed data that was excluded from the dimension object, and log object that will preserve all operations performed by the audit, dimension and quarantine objects. These mappings enable the preservation of variation, control, surrogate key and data fields between data objects, which are imposed by facts defined over both the SCD and the SCD-H signatures. For instance, for the *dimensionToHistory*, the correspondent mapping is established between a dimension object data as input and history object data as output. Additionally, several signature facts were used to enforce the correctness of each mapping, i.e., that the mapping associates fields of the same nature. This means that related fields are correctly mapped and that invalid relationships are avoided.

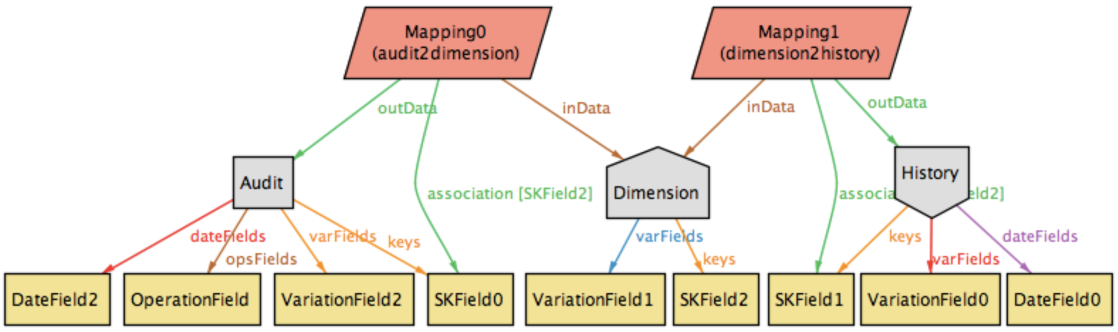

Figure 4. SCD pattern instance generated by the Alloy Analyzer

A predicate is then defined to embody the notion of consistent SCD: in this case, this amounts to checking whether the mappings of SCD are themselves consistent, as defined above. This property can then be automatically processed by the Alloy Analyzer, either to simulate instances that conform to the specification or to check for the correctness of concrete instances. Figure 4 denotes one such random, consistent, instance generated by the Alloy Analyzer for SCD patterns.

```
sig State {}
sig Value {}
sig Row {
    values: Field -> lone Value
}
abstract sig DataObject{
    (...)
    rows: Row -> State
}
fact rows {
    all s : State, o : DataObject, r : o.rows.s | r.values.Value = o.fields
}
pred addToDimension [s,s': State, r,r': Row, scd: SCD] {
    r in scd.auditToDimension.inData.rows.s
    r' not in scd.auditToDimension.outData.rows.s
    scd.auditToDimension.inData.rows.s' =
        scd.auditToDimension.inData.rows.s - r
    scd.auditToDimension.outData.rows.s' =
        scd.auditToDimension.outData.rows.s + r'
    all f : scd.auditToDimension.association.Field |
        f.(r.values) = f.(scd.auditToDimension.association).(r'.values)
    (...)
}
assert addToDimensionCorrect {
    all s: State, s': s.next, scd: SCD, r: Row |
        (consistentSCD[s,scd] and addToDimension[s,s',r,scd]) =>
            consistentSCD[s',scd]
}
check addToDimensionCorrect for 10
```
Figure 5. Excerpt of the Alloy specification for SCD-H pattern for dynamic behavior simulation

The specification presented in Figures 2 and 3 embodies the static constraints of ETL patterns. However, we are interested in checking the correctness of the process as data is collected from the sources and ETL procedures are executed. In Alloy, dynamic behaviour is encoded through well-established idioms, like the local state idiom followed in this work [3]. Roughly, a new signature *State* is introduced in the specification, that represents different states of the system, modelling the notion of evolving time. Artefacts that are expected to evolve in time are then appended with a *State* element, that will denote their value in each instant of time. In our scenario, only the data contained in each data object are expected to change (the structure of the data objects and the patterns is static). This dynamic version of the specification is presented in Figure 5, where a *Row* signature was introduced to describe the association between a *Field* and a specific *Value*. Data objects were then extended to contain a dynamic *rows* fields that denotes its data in each instant of time. An additional fact restricts rows to assign values to the fields of the parent data object.

The evolution of the system is then modelled through declarative predicates that relate two different states. In this perspective, if the static components of the specification represented the pattern configuration, these dynamic components define the pattern behaviour based on those configurations. Figure 5 depicts, as an example, predicate *addToDimension*, that models the behaviour of the insert operation of the pattern, involving the audit and target dimension objects. Being declarative, these definitions are usually comprised by pre-, post- and frame conditions. In this case, the pre-conditions restrict the operation to be applied only if the surrogate keys of the audit row that will be processed do not exist in the dimension table (otherwise the update operation should be applied instead); the post-conditions state that the row is removed from the audit table and inserted in the target dimension, preserving the data as defined by the associations of the SCD mappings; and the frame conditions state that every other artefact, like the history table, stay unchanged. The two states are represented by the *s* and *s'* parameters of the predicate: whenever the *rows* field is annexed with the *s* elements it represents the data in the pre-state, while *rows* at *s'* represents the data in the post-state.

Once these operation predicates are defined, the specification can be automatically analysed by the Alloy Analyser to check whether they preserve the consistency constraints defined above, within a bounded universe. These are defined in the *assert* clause in Figure 5, that tests whether the execution of the *addToDimension* operation preserves the consistency of the system. The *check* command effectively instructs the Alloy Analyzer to verify this assertion for a specific scope. For our specification, the Alloy Analyzer has shown that the three defined operations indeed preserve the specified consistency predicates, providing an increased level of confidence on the correctness of the procedure. Additional constraints and level of detail can be added to the specification to verify more complex properties.

Example

To provide a better picture of the application of SCD pattern and its configuration, let us consider a simple dimension: *Customer* that stores information about clients data for a given telecommunication company (Figure 6). Specifically, the customer personal data are stored as descriptive data and telecommunication plan details are represent a variation attribute, preserving history about the plan that each customer had in some point of time. The history dimension stores all plan changes from the beginning of customer activity with the company.

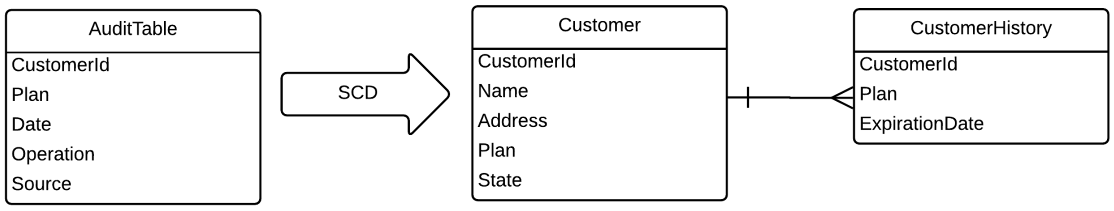

Figure 6. Case study data structures for the representation of a SCD-H scenario

Basically, the SCD pattern should be configured to create a specific instance that will extract data from *AuditTable* and according to the operation applied in source systems, trigger a specific action: the insert operation performs the insertion of the new row to the *Customer* dimension and its deletion from the audit table, the update operation results in the insertion of the changed record from the *Customer* dimension in the *CustomerHistory* dimension, using the *Date* attribute from audit table to set the *ExpirationDate* attribute, and the delete operation updates the state of the deleted record in dimension *Customer* to inactive. This example represents only a simple version of the problem, since no quarantine or log objects were considered.

Implementation

Based on the Alloy specification, a specific generator engine can be used to support the generation of specific instances through the use of the Alloy Analyzer, validating the constraints previously defined (similar approaches for different domains were already presented [23]) or following a test automation approach base on unit tests [24]. However, to support code generation, an easy and understandable language should be provided to simplify the configuration of each component. In previous works [25, 26], specific grammar components were describe to configure patterns metadata [25] using Xtext [27] framework. Thus, and based in the SCD-H pattern structure presented, Figure 7 presents a small excerpt of its configuration specifically for populating *Customer* and *CustomerHistory* dimension. The pattern configuration starts with an optional block that describes pattern description, using the *Name* and

Description keywords inside of the *Header* block. Blocks are identified using braces as delimiters. Next, four mandatory blocks are used: *input* to configure how and which data that should be read, *output* to describe where transformed data will be stored, *fields* that describe the fields that will be selected for the *Output* block, and the *options* block that describes the internal components associated to the SCD-H pattern. For *source* and *target* block, specific keywords are used to configured, respectively, source and target repository: the *data* keyword to identify the source name and *type* to identify the nature of data source used (in this case, a relational table is used). The *target* keyword identifies the target output in which data will be loaded. For this particular pattern, two output repositories should be configured: the dimension and historic dimension using specific blocks for each one of them with the same configuration structure. The *fields* block should be also configured according to each target repository, i.e. specific mapping fields should be provided enabling data operations between data objects. The mappings between repositories can be defined in one of the two blocks (in this case were defined in *dimension* block). Finally, the *options* block reveals specific options applied to SCD-H pattern, for example, the *key* to identify the field or fields that represents the SK, and the *operation* parameter that identifies the field or fields that stores details about operations performed for a specific record. All this parameters are related to audit table repository metadata. While the DSL structure and syntax can be validated using Xtext framework, the generated instances and respective behaviour can suffer from several problems. For example, the surrogate keys uniqueness after data insertion or referential integrity between dimension and historical dimension after process update can be validated before physical execution. Alloy can be used to check model consistency through a process that translates DSL primitives to Alloy primitives to check design errors. A automated approach is planned as future work.

```
use Transform.SCD.SCDH
   header{ name=SCD-H description = ...}
   source{
      data=AuditTable
      type=RELATIONAL
   }
   target{
      dimension{
        data=Customer
        type=RELATIONAL
      }
      history{
        data=CustomerHistory
        type=RELATIONAL
      }
   }
   fields{
     dimension{
      source.CustomerId = CustomerId
      source.Plan = Plan
      CustomerId=historic.CustomerId
      Plan=historic.Plan
     }
     history{
      source.Date = ExpirationDate
     }
   }
   options{
      keyField = CustomerId
      operationField = Operation
      (...)
   }
```

Figure 7. Excerpt of the SCD-H pattern configuration for the example provided in Figure 6

Based on this configuration, the domain-level instructions can be used for the generation of a specific instance following the specific language rules. For that purpose, some Java primitives can be used to implement code generators to a specific language or specific file structure that can be interpreted by some commercial tool. In our approach, a set of composed tasks (patterns) can be used to create the full ETL package. Several classes of patterns can be used to create a complete process from data extraction to data load, using a visual layer that simplifies process coordination. In past works [28], the BPMN was used to work as a visual layer for patterns representation. Next, the BPMN model is enriched using the proposed DSL, providing the necessary elements to enable the code generation for each pattern used. Thus, the DSL configuration for each pattern should be checked to verify its correctness and then the correspondent mapping processes to execution primitives can be triggered. Both Xtext and Alloy provides java API's that turn this process possible. The generation of code can be accomplished using a specific template-generator language, such as referred in [26].

4 Conclusions and Future Work

This paper proposed a pattern-oriented approach that allows the implementation of ETL processes with a higher level of abstraction. With the pattern-oriented approach presented, the knowledge and best practices revealed by several works can be put in practice using a set of software patterns that can be applied to the ETL development life cycle: from model representation to its physical implementation primitives. Such patterns can be used in software models, providing an excellent groundwork for process validation, allowing for the identification of the most important parts of a system to be built. A specific design pattern, the SCD-H, was identified along with its skeleton, representing its structure, constraints and behaviour according to specific rules. The presented approach keeps a specific template and instance as separated layers, since users need to provide data about data, i.e. users describe data repositories and its contents in structural terms and a specific generator will generate the respective code based on the primitives previously established and using specific data. The SCD-H pattern can be applied to any ETL scenario and to support this process, an excerpt of a formal specification in Alloy was presented, allowing pattern formalization through the use of the Alloy, which provides an automatic analysis that searches for false assertions through the generation of counterexamples. That way, patterns not only become easier to use than in the common granular approach, but can also be easily reused to produce better software, since models can be checked using a powerful simulation engine before its execution. This work represents a first attempt for formal specification using Alloy, representing static and behavioural specification for SCD-H. As future work, a more complete Alloy specification for ETL patterns is being developed, particularly in what concerns to behaviour formalization, assertion checking and exception/error handling scenarios that can occur. Additionally, several other patterns need to be formalized to provide a complete ETL package specification. A complete validation engine is also planned, translating the pattern configuration to Alloy models, allowing the ETL designers to seamlessly and automatically check the consistency of the developed patterns.

Acknowledgments

We like to express our thanks to Alcino Cunha, from HASLab R&D Centre, for his comments and suggestions done during the preparation of this paper.

References

1. Sommerville, I.: Software engineering (7th edition). (2004).
2. Kimball, R., Caserta, J.: The Data Warehouse ETL Toolkit: Practical Techniques for Extracting, Cleaning, Conforming, and Delivering Data. (2004).
3. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press (2012).
4. Köppen, V., Brüggemann, B., Berendt, B.: Designing Data Integration: The ETL Pattern Approach. Eur. J. Informatics Prof. XII, (2011).
5. Biggerstaff, T.J.: A perspective of generative reuse. Ann. Softw. Eng. 5, 169–226 (1998).
6. Vassiliadis, P., Simitsis, A., Georgantas, P., Terrovitis, M.: A framework for the design of ETL scenarios. In: Proceedings of the 15th international conference on Advanced information systems engineering. pp. 520–535. Springer-Verlag, Berlin, Heidelberg (2003).
7. Akkaoui, Z. El, Zimànyi, E., Mazón, J.-N., Trujillo, J.: A model-driven framework for ETL process development. Proc. ACM 14th Int. Work. Data Warehous. Ol. 45–52 (2011).
8. El Akkaoui, Z., Mazón, J.-N.N., Vaisman, A., Zimányi, E., Akkaoui, Z. El, Mazón, J.-N.N., Vaisman, A., Zimányi, E.: BPMN-Based Conceptual Modeling of ETL Processes. Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics). 7448, 1–14 (2012).
9. Akkaoui, Z. El, Zimanyi, E., Mazon, J.-N., Trujillo, J.: A BPMN-based design and maintenance framework

for ETL processes. Int. J. Data Warehous. Min. 9, 46 (2013).
10. Muñoz, L., Mazón, J.N., Trujillo, J.: A family of experiments to validate measures for UML activity diagrams of ETL processes in data warehouses. Inf. Softw. Technol. 52, 1188–1203 (2010).
11. Wilkinson, K., Simitsis, A., Castellanos, M., Dayal, U.: Leveraging business process models for ETL design. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). pp. 15–30 (2010).
12. El-Sappagh, S.H.A., Hendawi, A.M.A., El Bastawissy, A.H.: A proposed model for data warehouse ETL processes. J. King Saud Univ. - Comput. Inf. Sci. 23, 91–104 (2011).
13. Gabriel, R.P.: Patterns of Software Tales from the Software Community. Architecture. 239, (1996).
14. Appleton, B.: Patterns and Software: Essential Concepts and Terminology. Object Mag. Online. 2010, 1 (2000).
15. Aalst, W.M.P. Van Der, Hofstede, A.H.M. Ter, Kiepuszewski, B., Barros, A.P.: Workflow Patterns. Distrib. Parallel Databases. 14, 5–51 (2003).
16. Kimball, R., Ross, M.: The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling. Wiley (2002).
17. Santos, V., Belo, O.: No need to type slowly changing dimensions. In: IADIS International Conference Information Systems (2011).
18. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: On challenges of model transformation from UML to Alloy. Software and System Modeling 9(1), 69–86 (2010).
19. Kuhlmann, M., Gogolla, M.: From UML and OCL to relational logic and back. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). pp. 415–431 (2012).
20. Cunha, A., Garis, A., Riesco, D.: Translating between Alloy specifications and UML class diagrams annotated with OCL. Software and System Modeling 14(1), 5–25 (2013).
21. Büttner, F., Egea, M., Cabot, J., Gogolla, M.: Verification of ATL transformations using transformation models and model finders. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). pp. 198–213 (2012).
22. Macedo, N., Cunha, A.: Least-change bidirectional model transformation with QVT-R and ATL. Software and System Modeling (In press) (2015).
23. Khalek, S.A., Yang, G., Zhang, L., Marinov, D., Khurshid, S.: TestEra: A tool for testing Java programs using alloy specifications. 2011 26th IEEE/ACM Int. Conf. Autom. Softw. Eng. ASE 2011, Proc. 608–611 (2011).
24. Sullivan, A., Zaeem, R.N., Khurshid, S., Marinov, D.: Towards a Test Automation Framework for Alloy. In: Proceedings of the 2014 International SPIN Symposium on Model Checking of Software. pp. 113–116. ACM, New York, NY, USA (2014).
25. Oliveira, B., Belo, O.: A Domain-Specific Language for ETL Patterns Specification in Data Warehousing Systems. In: 17th Portuguese Conference on Artificial Intelligence (2015).
26. Oliveira, B., Santos, V., Gomes, C., Marques, R., Belo, O.: Conceptual-physical bridging - From BPMN models to physical implementations on kettle. In: CEUR Workshop Proceedings. pp. 55–59 (2015).
27. Foundation, E.: Xtext - Language Development Made Easy!, https://eclipse.org/Xtext/.
28. Oliveira, B., Belo, O.: ETL Standard Processes Modelling: A Novel BPMN Approach. In: 15th International Conference on Enterprise Information Systems (ICEIS) (2013).