Validating ETL Patterns Feasability using Alloy

Bruno Oliveira[1] and Orlando Belo[2]
[1]*CIICESI, School of Management and Technology, Porto Polytechnic, Felgueiras, Portugal*
[2]*ALGORITMI R&D Centre, University of Minho, Campus de Gualtar, 4710-057 Braga, Portugal*

Keywords: Data Warehousing Systems, ETL Patterns, ETL Component-reuse, Software Engineering, Formal Specification and Validation, BPMN, Alloy.

Abstract: The ETL processes can be seen as typical data-oriented workflows composed of dozens of granular tasks that are responsible for the integration of data coming from different data sources. They are one of the most important components of a data warehousing system, strongly influenced by the complexity of business requirements, their changing, and evolution. To facilitate the planning and ETL implementation, a set of patterns specially designed to map standard ETL procedures is presented. They provide a simpler and conceptual perspective that can enrich to enable the generation of execution primitives. Generic models can be built, simplifying process views and providing methods for carrying out the acquired expertise to new applications using well-proven practices. This work demonstrates the fundaments of an ETL pattern-based approach for ETL development, its configuration and validation trough a set of Alloy specifications used to express its structural constraints and behaviour.

1 INTRODUCTION

Even now, the development of data warehouses populating processes – ETL – remains a challenge in the area, engaging very diversified personnel with specific expertise and specific computational resources. Due to its complexity and heterogeneous nature, ETL systems development process addresses very challenging problems ranging from simple access to information sources to complex strategies for conciliating data and ensure their quality.

Usually, ETL architects and engineers start by initial drafts and models, getting a system overview with the main requirements that need to be validated before its implementation. Additionally, they need to know the data they are bringing from data sources as well establish the techniques applied in data extraction, transformation, and load phases. Abstract models are very useful even for initial versions since they have the ability to describe the system we want to implement, regardless of the methodology or technology used in its implementation. They revealing project needs in a very clear and precise way, representing "first picture" of the system, very useful for project discussion and planning (Losavio et al., 2001).

Abstract models can include the representation of software components that are frequently used for software modelling to describe common and well-known techniques that are used to describe specific system parts. The component reuse helps to produce software with better quality, faster and with lower costs since coarse grain components based on well-known design techniques are used. We believe that the pattern-oriented approach can be applied as well to ETL development, enabling its reuse in many different scenarios to solve recurring problems.

In this paper, we designed and developed a set of Alloy language specifications (Jackson, 2012) for expressing ETL patterns' structural constraints and behaviour. The Alloy is a declarative specification language that supports problem structural modelling and validation, helping to avoid process inconsistencies or architectural contradictions.

We selected one of the most relevant ETL patterns: The Data Conciliation and Integration (DCI), which has a set of operational requirements that are presented using the Alloy language. After this first introductory part, we present next to a briefly related work (section 2), and the ETL meta model designed for ETL patterns formalization with the DCI pattern specification as an example (section 3). The exception and log handling mechanisms that support each pattern operationally are also explored (Section

4), as well with a specific example applied to the DCI pattern (Section 5). Finally, in Section 6, we evaluate the work done so far, pointing out some research guidelines for future work.

2 RELATED WORK

After producing an initial plan using more abstract models, the selection of an ETL tool to support the system implementation appears quite naturally. Complex software pieces are usually supported by development processes covering all development stages: from requirements identification to the implementation and maintenance phases. The design task (conceptual and logical) supports the development team, identifying how the system should meet customer requirements and to determine how the system should be effective and efficient. The use of a design methodology to support ETL implementation using a specific tool was proposed by Vassiliadis and Simitsis (Vassiliadis et al., 2003, 2000). Several authors explored the ETL design phases using different approaches, languages and notations. The use of UML (Unified Modelling Language) (Trujillo and Luján-Mora, 2003) and BPMN (Business Process Model and Notation) (El Akkaoui and Zimanyi, 2009) are just two examples. The works presented by Munoz (Muñoz et al., 2009) and Akkaoui (El Akkaoui et al., 2011) revealed as very important contributions to our work, revelling important aspects to cover models translation to executable code, providing a way to describe the structure and semantic to produce final systems guided from more abstract models.

With this work, we propose a method to encapsulate patterns behaviour inside components that can be reused and mapped to more detailed primitives using a generator-based reuse approach (Biggerstaff, 1998). Based on pre-configured parameters, the generator produces a specific pattern instance that can represent the complete system or part of it, leaving physical details to further development phases. The use of software patterns to build ETL processes was first explored by Köppen (Köppen et al., 2011) that proposed a pattern-oriented approach to support ETL development, providing a general description for a set of patterns such as aggregator, history and duplicate elimination. The patterns are presented only at conceptual level, lacking to describe how they can be mapped to execution primitives. The approach presented in this paper works with high level components instead of using very granular tasks such as joins, union,

```
abstract sig Field{}
sig KeyField extends Field
sig PKField, SKField, FKField extends KeyField{}
sig ControlField, VariationField, DescriptiveField extends Field{}(...)
sig DataObject{fields: some Field,keys: some SKField}
fact dataObjectKeyFields {all o: DataObject | o.keys in o.fields(...)}
sig Mapping {inData: one DataObject,outData: one DataObject,association: Field
-> Field}
fact consistentMapping {
   all m: Mapping | m.association in m.inData.fields -> m.outData.fields
       (...)}
abstract sig PatternCore{sourceToTarget: some Mapping}
abstract sig Throwable{sourceToThrowable: some Mapping}
abstract sig Log{sourceToLog: some Mapping}
abstract sig Pattern{
   coreComponent:one PatternCore,
   throwableComponent: set Throwable,
   logComponent: set Log}
fact consistentPattern {(...)}
abstract sig Extract, Transform, Load extends PatternCore{}(...)
enum Rule{DELETE, PRESERVE}
sig IntegrationRules{associationFields : some Mapping,rule: one Rule,
   condition: one ConditionalOperator}
sig DCI extends Transform{rules: some IntegrationRules,compensationPattern:
lone Transform}
fact DCISameOutput{(...)}
```

Figure 1: Basic Alloy specification to support ETL patterns.

projections or selections. As well happens in other software areas, the use of abstraction techniques can be used to provide a simpler view over ETL conceptual models, and at the same time provide a flexible approach to enrich conceptual models in further stages. All this considering and using the efforts done in earlier stages. Thus, we believe that a formal model that describes model constraints and behaviour is needed to support the physical generation of ETL processes. We have been working in the last years in a pattern-oriented approach for ETL development (Belo et al., 2014; Oliveira et al., 2014). We already identified a set of ETL patterns that can be used to support all ETL development phases. Recently, we introduce the use of Alloy Language to formalize ETL patterns behaviour (Oliveira et al., 2016), showing the specification of one common ETL procedure: The Slowly Changing Dimension (SCD) pattern. In this work, we extend the work presented through the extension of pattern main components for exception and log handling, and with the DCI pattern specification.

3 ETL META MODEL FOR PATTERNS DEFINITON

The Model-driven techniques are being developed for several software areas and are used not only to support models development but also to cover code generation as well as model synchronization between physical models and abstract models. Whether being conceptual or logical, ETL models cover specific data requirements that reveal a symbiotic relationship between business requirements and process implementation. This means that each ETL is different even under the same domain, which compromises its reusability outside problem scope. To minimize such problems, an ETL framework covering a set of most common ETL procedures used is proposed.

The ETL pattern concept is represented by a "core" that encapsulates all pattern rules to support operational requirements and the logic behind it, the internal input and output interfaces to communicate both with ETL workflow and with the data layer to produce specific instances and the communication layer with other patterns. Additional ETL metadata is also preserved in the data layer, supporting the error and log strategies to handle errors and pattern events. Specific instances of "Throwable" and "Log" pattern components communicate with each associated pattern, encapsulating all logic behind error and log handling. The "Throwable" pattern component uses the input configuration to handle error or exception scenarios through the application of specific recovery strategies previously configured using pattern metadata. The unexpected scenarios that cause system critical failure can be configured to use specific procedures to maintain data consistency. For example, if some error compromises the ETL execution, the process can be aborted using a rollback procedure, maintaining data in a consistent state. The "Log" pattern stores ETL main events timeline to identify data lineage, bottlenecks, and errors. Thus, the ETL process can be analysed to identify error trends and to apply specific strategies to minimize them and eventually reduce ETL resources needed for subsequent loads.

In Figure 1, a set of formal specifications in Alloy is presented to express patterns structural constraints

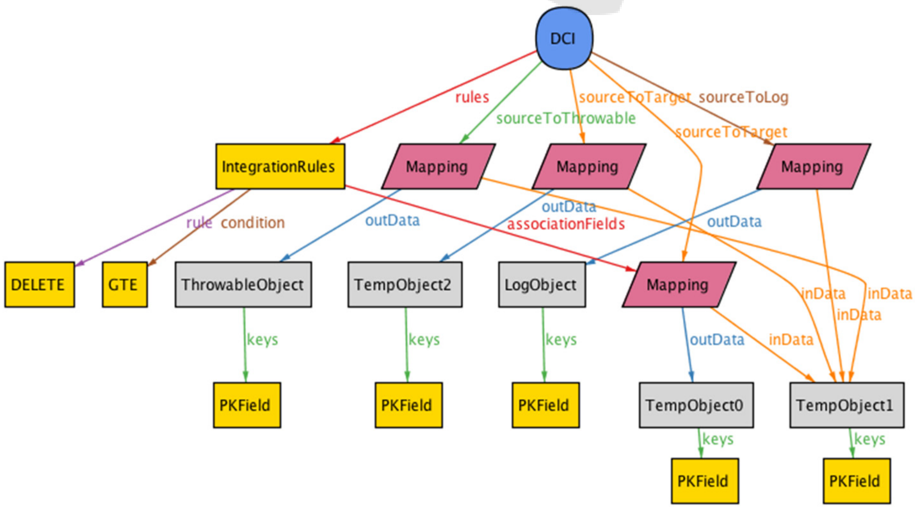

Figure 2: Pattern instance generated by the Alloy Engine Analyser.

and behaviour. The concepts hierarchy is 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, for example: “Field” representing patterns configuration properties and “DataObject” representing the type of data repositories related to the “Pattern” concept. The “Field” signature can be specialized in control fields (“ControlField”) to descriptive properties that store useful metadata related to each handled schema. For example: the date fields (“DateField”), describing temporal data; or the the log fields (“LogField”), describing specific metadata related to store the action performed during ETL execution.

The “PatternCore” signature represents the most general concept used, while the “Extraction”, “Transform” and “Load” signatures represent the three types of patterns that are intrinsically associated with each typical phase of an ETL process. The “Extraction” class instances are used to extract data from data sources using a specific data object (e.g. a table or file), representing typical extraction data processes and algorithms applied to specific data structures. The “Transformation” class represents patterns that are used in ETL transformation phase (Rahm and Do, 2000) to align source data schema requirements to the target DW schema requirements. This pattern category represents a large variety of procedures that are often applied in DWS, such as the application of data variation policies (SCD), generation of surrogate keys or schema transformations. Patterns of this class can also be specialized in concrete procedures considering its decomposition properties. For example, the “Data Quality Enhancement” (DQE) pattern can be specialized to the “Normalization” pattern that represents the set of tasks needed whenever it is necessary to standardize or correct data according to a given set of mapping rules. Thus, all the most frequent ETL patterns can be represented along with all its operational stages. Finally, the “Load” class represents patterns used to load data to the target DW repository, encapsulating efficient algorithms for data loading or index creation and maintenance. The “Intensive Data Loading” (IDL) signature represents the “Load” signature specialization, embodying the necessary operational requirements to load data to a target DW schema considering the schema restrictions followed.

The “Mapping” signature represents the association between fields, establishing the relationship between attributes from two different sources through the binary field association between fields (*a->b*). Additional constraints impose (“consistentMapping” fact), for example, that a mapping is only valid if it associated fields from the input (“inData”) and the output (“outData”) data sources. The “Pattern” signature is composed of a set of three different mappings (“Mapping” signature):

– “sourceToTarget”: describing the set of relationships between source and target fields for pattern application;
– “sourceToThrowable”: describing the set of relationships between the source fields and exception/error support fields;
– “sourceToLog”: describing the set of field relationships between the source schema and target log structures.

Each “PatternCore” specialization should preserve additional constraints over the arity of the fields to serve particular requirements. Signatures may contain fields of arbitrary arity that will embody the associations between the different artifacts. For example, the “DataObject” signature represent data repositories that contain a set of field declarations shared by its extensions: each “DataObject” element is related to a non-empty set of “Field” elements (imposed by the keyword “some”). A set of “SKField” elements ae also used to express the surrogate key fields.

The DCI pattern is a “Pattern” specialization that involves the integration of data about the same subject from several data source. Thus, several conflicts can occur that should be properly handled to provide consistent data view to reporting tools. Thus, it is necessary to integrate some related data (data from the same entity) coming from different data sources. The data integration should be accomplished using binding logic procedures that should be implemented to specify how related fields could be linked together. The Figure 1 presents an Alloy specification with the main concepts and relationships to support the DCI rules. The “DCI “signature represents the DCI pattern structure to support the logical mappings between each data source and target data objects for a given configuration. Several “sourceToTarget” mappings can be provided, however, the output data object should be only one, which is forced by the “DCISameOutput” fact. This means that for each object that we need to integrate data, a specific DCI instance should be used. For that, specific integration rules (“IntegrationRules” signature) should be provided to identify the action that should be applied to each field association. Thus, when multiple sources are used, conflicts can be avoided using specific rules ("Rule" enumeration) to decide which fields from a specific data object are used to populate the target DW object. The "consistentDCI" fact enforces several constraints to guarantee the DCI consistency. For example, the fields associations used for rules configuration should be defined in "sourceToTarget" mappings. A predicate is then defined to embody the notion of consistent DCI: in this case, checking whether the mappings of DCI are themselves consistent according to the rules defined above. This property can then be automatically processed by the Alloy Analyser, either to simulate instances that conform to the specification or to check for the correctness of concrete instances. The Figure 2 presents a random, consistent, instance, composed by two "sourceToTarget" mappings and one "sourceToLog" and "sourceToThrowable" mapping.

4 THE THROWABLE AND LOG PATTERNS

The ETL implementation and management deal with several problems that can compromise all process activities. Data coming from operational systems can suffer from inconsistencies related to bad data input or even from changes performed to cover new business needs. Using specific "throwable" components for each pattern instance allows for the definition of specific strategies to plan and deal with recurring problems, anticipating them and provide suitable recovery strategies. However, stopping all process every time an unexpected situation occurs can be impractical. For that reason, a more flexible approach should be followed to prevent process failure, providing a way to separate inconsistent records from ETL normal flow or, if possible, correct them at execution time.

The Figure 3 represents the main concepts related to the "throwable" pattern using Alloy primitives. The "Exception" signature represent data errors that do not compromise ETL execution and for that reason, can be handled or avoided using specific routines. Incoherent records are moved to quarantine ("Quarantine" signature) data objects, whose structure is determined according to each pattern configuration. The quarantine meta data structure is generally composed by a "Key" attribute identifying each quarantine row and a set of control attributes ("controlAtt") that are used to tag each row with additional metadata that can be provided by the exception handler mechanisms. The temporal data ("Temporal" signature), the source object ("objectDescription") where the exceptions are found, the triggered exception ("Throwable"), the action that triggered the exception ("event"), the evaluation state ("State", by default 'invalid') and the exception severity ("severityScore") are some examples of possible control fields used to store errors metadata. Additionally, descriptive attributes are used to represent non-structured data to enrich the quarantine objects with information that can be used for human interpretation, while the record identification ("recordId") is used to identify source records stored in quarantine object.

```
(...)
enum SeverityCode{LOW, MEDIUM, HIGH}
sig ThrowableField, StageField extends Field{}
sig ThrowableMetaData {
   temporalData:some DateField, objectDescription:one Field, (...)}
sig QuarantineData extends DataObject{metaData:one ThrowableMetaData & fields}
abstract sig Throwable extends Pattern{sourceToThrowable:some Mapping, log:one
LogPattern}
sig Exception extends Throwable{quarantineObject:one
QuarantineData,correctionProcedure:one DQE}
sig Error extends Throwable{metadata:one ThrowableMetaData}(...)
-- Log specification
abstract sig LogField extends Field{}
abstract sig EventLogField extends LogField{}
sig CompensationField, ErrorLogField, CheckPointField extends EventLogField{}
sig SequenceField, PerformanceField extends LogEventField{}
enum LogLevel{EXCEPTION, FATAL, INFO}
sig LogObject extends DataObject{temporalData:some DateField & fields,(...)}
abstract sig Log extends Pattern{sourceToLog: some Mapping}(...)
```

Figure 3: Alloy specification describing "throwable" and "Log" patterns.

The “Error” class represents serious data errors that cannot be handled automatically, compromising all ETL flow. When errors are detected, rollback mechanisms can be configured to reverse all operations done so far and return data to a consistent state. Otherwise, the process can be cancelled, leaving data at the current state. Additionally, to the ability to track error or unexpected situations, the ETL process must detect data inconsistent states to avoid its progression to DW system. The transaction logs files are a common strategy used by many software systems to collect data about the events/transactions that occurred during system daily operations. All the performed actions can be stored to provide a system picture in a specific time-window. For that, critical tasks should be monitored to ensure quality/performance measures, identifying bottlenecks and errors that can influence the DW operationally. For example, log mechanisms can be used to control the performance or quality measures for specific tasks and check process consistency based on the execution history. Strong variations in the results can point to source systems anomalies that should be investigated.

The log structure can also be used to support recovery mechanisms that are used to protect ETL procedures against critical failures, providing a way to rollback ETL data to a consistent state. The “Log” pattern is composed by specific mappings (“sourceToLog”) that relate the field association between the source object and target log Object (“LogObject”). The log objects use specific fields to check quality parameters such as the temporal (“temporalData”) and performance data (“taskDescription” using specific “performanceFields”) about process tasks. The number of records processed and the time needed to perform particular tasks are just some example of possible performance fields that can be used. The log level (“LogLevel” enumeration) describes the type of log used. For example, information log can represent its entries as information logs (‘INFO’), which are useful to describe informational messages that highlight progress tasks, or fatal logs (‘FATAL’) that describe critical error events that lead the process to abort. The log description (“logDescription”) describes the event that originates the log entry, which can be an exception, error or a state indicating task finishing. Finally, the key referencing the source object and the sequence of log entries (primary key) can be described using “sequence” fields.

5 A DCI PATTERN EXAMPLE

The Figure 4 presents a BPMN algorithm with the

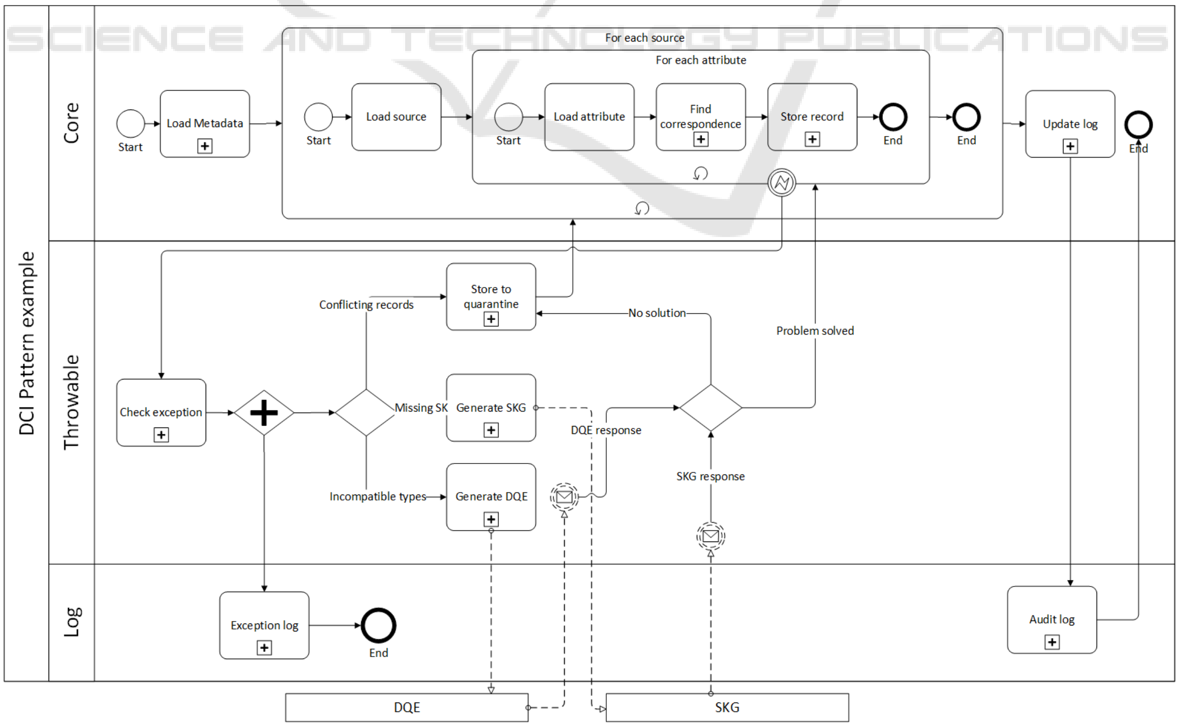

Figure 4: BPMN example algorithm for DCI instantiation.

main tasks to support the DW data load process, involving a "Product" dimension. Considering that several data sources are used to populate the "Product" dimension without the guaranty of well-conformed data, a specific DCI instance was configured to deal with such scenario. The BPMN process describes two cyclic structures to handle each input source ("Load source" sub process) and the records they bring, respectively. For each record ("load record" sub process), the mappings that preserve the binding rules (derived from "sourceToTarget" Mapping) must be used to identified to enable the integration logic. Thus, for each processed record, a specific lookup process over the mapping table is performed to find the correspondent plan name ("Find correspondence" sub process). When records are stored ("Store record" sub process), specific events can be triggered to identify three possible exception scenarios ("Throwable" pool):

– "Missing SK" referring to the new records without correspondent SK;
– "Incompatible types" for data type incompatibilities between the product id attribute from source and target repositories;
– "Contradictory records" for product names with contradictory values.

While for the contradictory values, records are stored in quarantine objects using "Store quarantine" sub process, for the "Missing SK" and "Incompatible types" exceptions, specific Surrogate Key Generator ("Generate SKG" sub process) pattern instance and DQE instances ("Generate DQE" sub process) are generated to deal with each scenario. The logic between each pattern handling is represented using collapsed BPMN pools ("SKG" and "DQE", respectively). Then, "Throwable" instances, specific resources can be instantiated and invoked to correct wrong values before proceeding to its storage. When the conflict is successfully managed, records are reintroduced to regular ETL flow. If the conflict persists, the quarantine table is updated with all conflicting records (a BPMN conditional gateway is used to select the right path) and a specific log entry is registered.

The "Log" pool presents two configured scenarios, representing the "Exception log" and the "Audit log" sub processes. The logic behind each one is straightforward, consisting of the identification of several control attributes to identify complementary data about the actions performed in the remaining BPMN lanes and its storage according to its purpose: the "Exception Log" stores "Throwable" events while the "Audit Log" stores checkpoint events triggered when the populating processes reaches a specific milestone. In this case, the "throwable" configuration was configured with a log component ("log" field), allowing the communication between the components. That way, when "throwable" events are triggered, the exception and error logs can be properly updated.

The DCI pattern instance presented was generated based on the specification principles presented in Figure 1 and Figure 3. 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 (Khalek et al., 2011) or following a test automation approach (Sullivan et al., 2014). However, to support code generation, an easy and understandable language should be provided to simplify the configuration of each component. In previous works (Oliveira et al., 2015; Oliveira and Belo, 2015), specific grammar components were described to configure patterns metadata. An automated approach is planned to use the Alloy can be used to check model consistency through a process that translates grammar primitives to Alloy primitives to check design errors. Based on this configuration, code generators to a specific language or specific file structure that can be interpreted by a specific commercial tool can be used to create the full ETL package.

6 CONCLUSIONS AND FUTURE WORK

Because of the existence of more data and more complex business processes, the data processing demands have increasing the ETL development complexity has been studied with the goal to simplify its development and reduce the potential risks to its implementation. The efforts did so far help in the identification of recurrent problems and respective strategies to solve them. Still, ETL systems are considered very time-consuming, error-prone and complex systems since each DW deals with their own data with specific requirements. With the use of Patterns, these strategies can be encapsulated and parameterized, providing a powerful groundwork for process validation and allowing for the identification of the most important parts of a system to be built.

The Pattern-oriented approach presented relies on the use of software components that represent a template of how a specific problem can be solved. To formalize the pattern structure as well its operational constraints, a specific Alloy model is proposed to guarantee some level of confidence for the generation of physical instances through the use of a simulation engine that searches for instances representing false assertions according to a specific set of conditions. Thus, since models can be checked before its execution, a new integrity level is sustained, ensuring that pattern structure is consistent with their specification. The DCI pattern, was presented along with its skeleton, keeping a specific template and instance as separated layers. For pattern instantiation, the physical objects should be described at structural terms. Thus, a specific generator is being developed (Oliveira et al., 2015) to generate the respective code based on the primitives previously established

The presented specification only covers the patterns static representation, however, as future work, we intend to enrich this specification with behavioural specification, covering the main operations and states related to each pattern application. We intend to enrich this specification with behavioural specification and assertion checking based on the main pattern components and states, including the "Throwable" and "Log" components configuration.

REFERENCES

Belo, O., Cuzzocrea, A., Oliveira, B., 2014. Modeling and Supporting ETL Processes via a Pattern-Oriented, Task-Reusable Framework. In: *IEEE 26th International Conference on Tools with Artificial Intelligence*.

Biggerstaff, T.J., 1998. A perspective of generative reuse. *Ann. Softw. Eng. 5*, 169–226.

El Akkaoui, Z., Zimanyi, E., 2009. Defining ETL worfklows using BPMN and BPEL. In: *Proceeding of the ACM Twelfth International Workshop on Data Warehousing and OLAP DOLAP 09*. pp. 41–48.

El Akkaoui, Z., Zimànyi, E., Mazón, J.-N., Trujillo, J., 2011. A Model-driven Framework for ETL Process Development. In: *Proceedings of the ACM 14th International Workshop on Data Warehousing and OLAP, DOLAP '11*. ACM, New York, NY, USA, pp. 45–52.

Jackson, D., 2012. Software Abstractions: Logic, Language, and Analysis. *MIT Press*.

Khalek, S.A., Yang, G., Zhang, L., Marinov, D., Khurshid, S., 2011. TestEra: A tool for testing Java programs using alloy specifications. 2011 *26th IEEE/ACM Int. Conf. Autom. Softw. Eng. ASE 2011, Proc.* 608–611.

Köppen, V., Brüggemann, B., Berendt, B., 2011. Designing Data Integration: The ETL Pattern Approach. *Eur. J. Informatics Prof. XII*, 49–55.

Losavio, F., Chirinos, L., Pérez, M.A., 2001. Quality Models to Design Software Architectures. In: Technology of Object-Oriented Languages and Systems. TOOLS 38. *IEEE Computer Society*, Zurich, pp. 123–135.

Muñoz, L., Mazón, J.-N., Trujillo, J., 2009. Automatic Generation of ETL Processes from Conceptual Models. In: *Proceedings of the ACM Twelfth International Workshop on Data Warehousing and OLAP, DOLAP '09. ACM*, New York, pp. 33–40.

Oliveira, B., Belo, O., 2015. A Domain-Specific Language for ETL Patterns Specification in Data Warehousing Systems. In: Springer (Ed.), *17th Portuguese Conference on Artificial Intelligence (EPIA'2015)*. Coimbra, pp. 597–602.

Oliveira, B., Belo, O., Cuzzocrea, A., 2014. A pattern-oriented approach for supporting ETL conceptual modelling and its YAWL-based implementation. *3rd Int. Conf. Data Manag. Technol. Appl. DATA 2014* 408–415.

Oliveira, B., Belo, O., Macedo, N., 2016. Towards a Formal Validation of ETL Patterns Behaviour. In: Bellatreche, L., Pastor, Ó., Almendros Jiménez, J.M., Aït-Ameur, Y. (Eds.), Model and Data Engineering: 6th International Conference, MEDI 2016, Almer{í}a, Spain, September 21-23, 2016, Proceedings. *Springer International Publishing*, Cham, pp. 156–165.

Oliveira, B., Santos, V., Gomes, C., Marques, R., Belo, O., 2015. Conceptual-physical bridging - From BPMN models to physical implementations on kettle. In: *CEUR Workshop Proceedings*. pp. 55–59.

Rahm, E., Do, H., 2000. Data cleaning: Problems and current approaches. *IEEE Data Eng. Bull. 23*, 3–13.

Sullivan, A., Zaeem, R.N., Khurshid, S., Marinov, D., 2014. Towards a Test Automation Framework for Alloy. In: *Proceedings of the 2014 International SPIN Symposium on Model Checking of Software, SPIN 2014. ACM*, New York, NY, USA, pp. 113–116.

Trujillo, J., Luján-Mora, S., 2003. A UML based approach for modeling ETL processes in data warehouses. In: *International Conference on Conceptual Modeling*. pp. 307–320.

Vassiliadis, P., Simitsis, A., Georgantas, P., Terrovitis, M., 2003. A framework for the design of ETL scenarios. In: *Proceedings of the 15th International Conference on Advanced Information Systems Engineering, CAiSE'03*. Springer-Verlag, Berlin, Heidelberg, pp. 520–535.

Vassiliadis, P., Vagena, Z., Skiadopoulos, S., Karayannidis, N., Sellis, T., 2000. ARKTOS: A tool for data cleaning and transformation in data warehouse environments. *IEEE Data Eng. Bull 23*, 42–47.