Cheptre solution to the TTC 2023 incremental Class2Relational case

Frédéric Jouault[1,2], Nicolas Pouillard[3]

[1]University of Angers, LERIA, 49000 Angers, France
[2]ESEO-TECH / ERIS, 49100 Angers, France
[3]STACK'S, Angers, France

Abstract
This paper presents a batch solution to the TTC 2023 "Incremental MTL vs. GPLs: Class into Relational Database Schema" case. This solution is expressed in a domain-specific language that is not a model transformation language. It is therefore not one of the kinds of solutions that were expected. However, we believe that valuable insights may be gained from this experiment.

Keywords
model transformation, linear logic, Class2Relational

1. Introduction

The "Incremental MTL vs. GPLs: Class into Relational Database Schema" case [1] from the 2023 edition of the Transformation Tool Contest (TTC) asks for solutions written in either a Model Transformation Language (MTL), or a General Purpose Language (GPL). A MTL is a special kind of Domain-Specific Language (DSL) intended to write model transformation programs. As such, it typically offers higher-level constructs, such as model transformation rules, than typically found in a GPL, such as Java. However, it seems that it should also be possible to write a solution in a non-MTL DSL, for instance if it is Turing complete, or if it at least provides some minimal facilities. This paper investigates this idea by describing work on a solution written in a variant of the Ceptre [2] DSL. This language is typically used to model generative interactive systems, but we use it here to write a model transformation. The presented solution is not yet incremental, but is already able to perform batch computations. We believe that it should be possible to create an incremental solution in this language. The batch solution could then be used as a baseline to evaluate how much more complexity is required for an incremental one. This paper is organized as follows: Section 2 gives a brief overview of Ceptre, and the variant we used, called Cheptre. An overview of the solution is presented in Section 3. Finally, Section 6 gives some concluding remarks.

TTC'23: 15th Transformation Tool Contest, Part of the Software Technologies: Applications and Foundations (STAF) federated conferences, Eds. A. Boronat, A. García-Domínguez, and G. Hinkel, 20 July 2023, Leicester, UK.
frederic.jouault@eseo.fr (F. Jouault); np@nicolaspouillard.fr (N. Pouillard)

2. Ceptre/cheptre overview

The Ceptre language is a logic programming language [3] based on linear logic [4]. Ceptre's syntax is textual, and was created to model generative interactive systems, such as games and dynamic stories. The variant of Ceptre we use here is called Cheptre, and offers a few extensions. The state of the system is called its *context*, and is basically a multi-set of predicates. These predicates are considered as the truth currently holding. The programmer specifies rules to define how this context may evolve over time. These rules follow the principles of linear logic: they may consume or create predicates. For instance, the following rule specifies that two coins are necessary to buy an apple:

```
buy-apple: coin * coin -o apple.
```

Ceptre identifiers may contain hyphens ("`-`"). An optional name may be given to a rule by placing it at its beginning, and separating it with a colon. Here, `buy-apple` is the name of the rule. The `-o` operator, which reads as *lolli*, an abbreviation of lollipop, because of its shape, corresponds to linear logic implication. Each rule has a left-hand side (LHS), and a right-hand side (RHS). An empty LHS or RHS is denoted as `()`. The `*` operator, which reads as *tensor*, is used to join multiple predicates. In the above example, `coin`, and `apple` are two predicates. To be read as truth statements, they can be respectively read as "I own a coin" and "I own an apple". Notice the stark difference with the classical variation of this statement: "If (I own a coin) and (I own a coin) then (I own an apple)" in which you keep your coins and get an apple.

Rules can be grouped into stages. There is exactly one stage in the context, which denotes the current stage. Only rules that are within the current stage, or that go out of it, are available for matching at any given time.

A rule is matched, and therefore fireable, if its LHS is matched. Matching `a * b` consists in matching both `a` and `b`. Matching a predicate consists in finding it in the context. When a rule is fired, the predicates it matched are removed from the context, and the predicates that are specified on its RHS are added to the context. When, in a given context, there are no fireable rules, quiescence is reached. A special action is then automatically triggered and enables to continue. In Ceptre, quiescence adds to the context the special predicate `qui`. This makes it possible to specify rules that become fireable once a stage can no longer make progress.

The "$" modifier may be applied to LHS predicates, in order to specify that they must be matched but not consumed. The Cheptre extension of Ceptre also adds the "?" modifier, which may be applied to RHS predicates, in order to specify that they must be added to the context only if they are not already present in it (otherwise they would be duplicated). For instance, in the following rule, the agent (represented by variable `A`) must be at the store to buy an apple, but buying an apple will not change the agent's location. Moreover, after buying an apple, the agent is remembered as an buyer if that was not already the case, but buying multiple apples will not result in multiple instances of `buyer A` to be in the context.

```
$at A store * coin * coin -o apple * ?buyer A.
```

Without these modifiers one would need to: restore the agent location in the RHS, and also write another rule to remove duplicate `buyer` predicates, such as:

```
at A store * coin * coin -o at A store * apple * buyer A.
merge-buyer-duplicates: buyer A * buyer A -o buyer A.
```

Predicates can contain data structures. Additionally, complex navigation over the context, and these data structures, can be specified using a backward-chaining Prolog [5]-like language construct called `bwd` relations.

Constructor names for types, data, predicates, `bwd` relations, and rules must start with a lower case character. Variable names start with an upper case character. Atoms are another addition specific to Cheptre, inspired by αProlog [6]. Atoms are strings with an interface limited to equality. They start with a single quote (e.g., `'abc` or `'Test142`).

The Ceptre language being typed, it offers mechanisms to define types and give type signatures to data constructors, predicates, and `bwd` relations.

Cheptre extends Ceptre with built-in functions, two of them being used in this solution. `try` can be used to test if a predicate or `bwd` relation holds or not. Using our previous example, we could use `try` to emulate the `?` modifier. The first rule can only fire when the agent is already a buyer, and the second one can only fire when the agent is *not* a buyer.

```
$at A store * coin * coin * $buyer A -o apple.
$at A store * coin * coin * try (buyer A) == false
    -o apple * buyer A.
```

Another built-in function used in this solution is `fmt`, which performs string interpolation, and is used in the serialization step. Finally, this solution uses the built-in predicate called `fresh`. Used in a rule, it ensures that an atom has never been used before.

3. Solution overview

We implemented the transformation itself as a set of rules contained in a single stage named *apply*. However, we embedded this transformation stage into a multi-stage Cheptre program, in order to be able to separate model loading, transformation, and model serialization phases. Figure 1 represents all stages as boxes, and rules that change stages, which we will call transitions below, as arrows. Rule names are listed in each box below the stage name. Transitions between stages are labeled with their LHS and RHS, which are all empty (i.e., `()`) here. This Cheptre program is launched by a Java driver.

The Java driver starts the Cheptre program in the *idle* stage. While in this stage, the Cheptre pogram waits for the transition from *idle* to *apply* to be explicitly fired. The Java driver then sets the context to be the source model. This requires a conversion, implemented in Java, from XMI (loaded with EMF) to the Cheptre syntax. Then the Java driver fires the transition from *idle* to *apply*. All transformation rules are then applied until no rule can be applied any more. The Cheptre program then automatically transitions to stage *wait-before-serialization*, and waits for the transition to *erase-in* to be executed. The next step consists in the Java driver firing this transition, which results in all the rest of the program to be automatically executed. Once this has completed, the context contains an output predicate encapsulating an almost-XMI serialization of the output model. It is not necessary to implement XMI serialization in Cheptre, because it could have been implemented in Java. However, having it in Cheptre may prove useful, if at least to provide more rule and stage examples. Finally, the Java driver post-processes the Cheptre-generated XMI before serializing it. The post-processing is necessary to implement the firstToLower helper, which is currently not implementable in Cheptre, which lacks the appropriate string manipulation primitives. It also removes some artefacts that would render the XMI invalid.

Stage *apply2* contains a variant of the transformation, more fine-grained, which should be a better starting point for an incremental solution. It is currently not used by default, but it is an example of an alternative way to express model transformations in Cheptre. Moreover, it can be relatively easily enabled by changing the transition from stage *idle* to target stage *apply2* instead of stage *apply*.

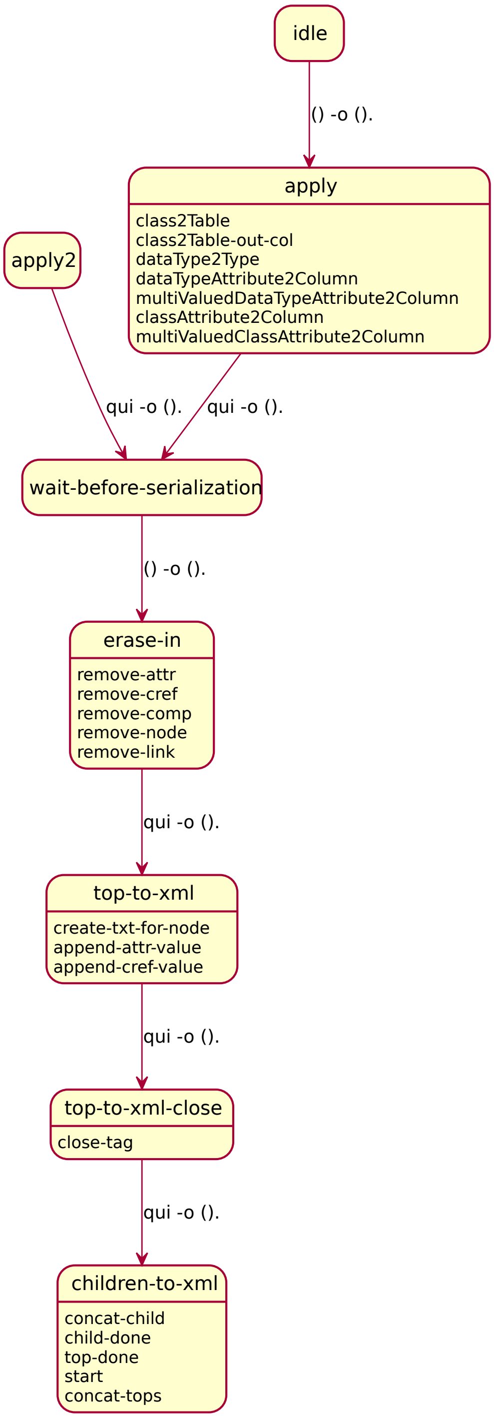


Figure 1: Cheptre solution stages

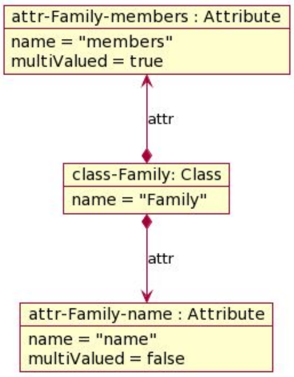
4. Model representation

We use four predicates to encode models:

- The *node* predicate represents a model element, and takes three arguments. The first argument is an atom identifying the model to which the element belongs. We use *'IN* for the source model, and *'OUT* for the target model. The second argument is an atom identifying the meta-class that types the model element (e.g., *'DataType* for the DataType meta-class of the Class metamodel). Finally, the third argument is an atom that corresponds to the identifier of the model element.
- The *attr* predicate represents attribute values, and takes three arguments. The first argument is an atom identifying the model element for which this predicate specifies an attribute value. The second argument is an atom identifying the feature for which this predicate specifies a value (e.g., *'name* for the name attribute of the NamedElt meta-class, or of the Named meta-class). The third argument corresponds to the value. Values may be strings (e.g., `(str "a string")`), booleans (e.g., `(boolean true)`) or null (i.e., `nullv`).
- The *cref* predicate represents cross-references (i.e., references that are not compositions), and takes three arguments. The first two arguments are the same as for the *attr* predicate. The third and last argument corresponds to the id of the model element that is the target of the reference.
- The *comp* predicate represents compositions (i.e., references that are containments), and takes three arguments. These arguments are the same as for the *cref* predicate.

The metamodels themselves are not represented, because they are not necessary to solve the Class2Relational case. Only meta-class and meta-class feature identifiers are used. This is different from the model representation presented in [7], which aims at proposing a formal definition of models.

Figure 2 shows how an excerpt of a Class model is represented. On the left-hand side, an object diagram is used to represent the model to be encoded into predicate applications. Each model element is represented as an object: a single class named Family owning two attributes named members, and name. The first attribute is multi-valued, whereas the second one is not. On the right-hand side, the corresponding Cheptre encoding is given. It consists of a comma-separated list of applied predicates. There is one application of the *node* predicate for each model element: for the members Attribute, the Family Class, and the name Attribute. The first argument is the model identifier: *'IN*, the second is the meta-class




```
node 'IN 'Attribute 'attr-Family-members,
attr 'attr-Family-members 'name (str "members"),
attr 'attr-Family-members 'multiValued (boolean true),
%cref 'attr-Family-members 'type 'class-Person,

comp 'class-Family 'attr 'attr-Family-members,

node 'IN 'Class 'class-Family,
attr 'class-Family 'name (str "Family"),

comp 'class-Family 'attr 'attr-Family-name,

node 'IN 'Attribute 'attr-Family-name,
attr 'attr-Family-name 'name (str "name"),
attr 'attr-Family-name 'multiValued (boolean false),
%cref 'attr-Family-name 'type 'type-String,
```

Figure 2: Model representation example

identifier, and the last is the element identifier. There is also one application of the *attr* predicate with *'name* as feature identifier, and the element's name as value, for each element. The two attributes also have one application each of the *attr* predicate with *'multiValued* as feature identifier. One application of the *comp* predicate is used to represent the composition from the class to each attribute. Finally, two commented out applications of the *cref* predicate are given to show how the type of each attribute would be represented in a more complete model.

Additionally, the transformation makes use of the *link* predicate to represent trace links. It takes two arguments: the identifiers of a source, and of a target model elements.

5. Transformation rules

In stage *apply*, the transformation rules basically correspond to the equivalent ATL rules (one of the reference solutions provided with the case), and are given the same names, but starting with a lower case. We refer to this approach as *coarse-grained*, because a single rule application adds a whole set of predicates to the context. In stage *apply2*, there is one Cheptre rule per ATL rule target element (named `<ATL rule name starting with lower case>-<ATL target element name>`) and one Cheptre rule per ATL binding (named `<ATL rule name starting with lower case>-<ATL target element name>-<target feature name>`). We refer to this approach as *fine-grained*, because a single rule application adds a single target model predicate to the context, plus a trace link for target element rules. Section 5.1 presents the coarse-grained approach, whereas Section 5.2 presents the fine-grained approach. Finally, Section 5.3 discusses some perspectives regarding incremental execution of these rules.

5.1. Coarse-grained variant

The rationale for this approach is that it closely follows the ATL reference solution's structure. There is one extra rule, when compared to the ATL solution: *class2Table-out-col*, which is responsible for attaching columns generated from single-valued attributes to the appropriate target table. This must be specified separately from the *class2Table* rule, because no iteration mechanism can be used within a rule in Cheptre. Therefore, it must be replaced with multiple applications of a single rule. Variable names were chosen to be the same as in the ATL transformation, except that they are in upper case (for source elements), or atoms (for target elements).

Listing 1 gives the code of the simplest rule: *dataType2Type*, which we use here as an example. Its LHS starts by matching the source element with the *node* predicate (without consuming it, hence the $), storing its identifier in the DT variable. It then matches the source element's name *attr* (without consuming it), storing its value in variable Name. The *fresh* built-in predicate is then used to create a new atom, guaranteed to be distinct from any others (i.e., a unique identifier), which is referenced as *'out* within the scope of this rule. Finally, the *try* built-in function is used to check that there is not already a trace link for DT, otherwise the rule does not match.

The RHS of rule *dataType2Type* starts by creating a target element of type *'Type*, by adding an instance of

Listing 1: Coarse-grained rule *dataType2Type* from stage *apply* that transforms a source DataType into a target Type

```
dataType2Type
  : $node 'IN 'DataType DT
  * $attr DT 'name Name
  *  fresh 'out
  *  try (link DT _) == false
 -o  node 'OUT 'Type 'out
  *  attr 'out 'name Name
  *  link DT 'out
  .
```

node to the context, with unique identifier *'out*. It then adds an instance of *attr* to the context, giving it the same value as the source element's. Finally, it adds a trace link between the source and target elements to the context.

There are multiple ways to express model transformations in Cheptre. Our solution is not tuned for performance.

Regarding syntactic complexity annotations: we only annotated the apply stage, because the other stages are not part of the transformation itself.

5.2. Fine-grained variant

The rationale for this approach is that it is closer to an incremental solution than the fine-grained variant. Listing 2 gives the two Cheptre rules that correspond to the *DataType2Type* ATL rule. The first one (*dataType2Type-out*) is responsible for matching a data type, and creating the corresponding target *node*. In contrast with rule *dataType2Type* from Listing 1, it does not create the *attr* application that would specify the target element's name. This also means it does not need to match the source element's name. However, it still needs to create a trace link. Instead of the *link* predicate used in the coarse-grained approach, the *nlink* predicate is used instead. It takes an additional argument (the second one), which identifies the rule that created the target element. This is necessary when a single reference ATL rule creates multiple target elements, because there is then one fine-grained rule for each of these target elements, with each requiring a separate trace link. Rule *dataType2Type-out-name* matches the trace link created by rule *dataType2Type-out*, and copies the source element's name into the target element's name.

5.3. Incrementality perspectives

Although neither of these variants support incrementality, the second one is closer to being able to do so. If new elements were added to the source model, Cheptre would be able to match and transform them. This corresponds to a form of append-only incrementality. This would of course require extending the Java driver to support this. In the coarse-grained variant, it would only be able to transform whole elements at once, for instance a data type, along with its name. However, in practice, a new data type may be first created, and its name may be set at a later time. The coarse-grained variant would not be able to do anything with the unnamed data type, which would delay change propagation until it is named. Conversely, the fine-grained variant would be able to transform the unnamed data type as soon as it is available, using rule *dataType2Type-out*. Its name could then be translated by rule *dataType2Type-out-name* when it becomes available. Full support for incrementality would require additional rules to remove target elements when source elements disappear.

Listing 2: Fine-grained rules *dataType2Type-out* and *dataType2Type-out-name* from stage *apply2* that transform a source DataType into a target Type

```
% One rule per target pattern element
dataType2Type-out
  : $node 'IN 'DataType DT
  *  try (nlink DT
       'DataType2Type-out _) == false
  *  fresh 'out
 -o  node 'OUT 'Type 'out
  *  nlink DT
       'DataType2Type-out 'out
  .

% One rule per binding
dataType2Type-out-name
  : $nlink DT
       'DataType2Type-out OUT
  * $attr DT 'name Name
 -o ?attr OUT 'name Name
  .
```

6. Conclusion

This paper has presented a Cheptre solution to the "Incremental MTL vs. GPLs: Class into Relational Database Schema" TTC 2023 case. This solution is not (yet) incremental. However, it provides an example of using a non-MTL DSL to implement a model transformation. Moreover, we believe that we will be able to make it incremental, in time.

References

[1] S. Greiner, S. Höppner, F. Jouault, T. L. Calvar, M. Clavreul, Incremental mtl vs. gpls: Class into relational database schema, in: Proceedings of the 15th Transformation Tool Contest (TTC 2023), 2023.

[2] C. Martens, Ceptre: A language for modeling generative interactive systems, Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment 11 (2015) 51–57. URL: https://ojs.aaai.org/index.php/AIIDE/article/view/12784. doi:10.1609/aiide.v11i1.12784.

[3] K. R. Apt, Logic programming., Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B) 1990 (1990) 493–574.

[4] J.-Y. Girard, Linear logic: Its syntax and semantics, in: Proceedings of the Workshop on Advances in Linear Logic, Cambridge University Press, USA, 1995, p. 1–42.

[5] I. ISO, I. ISO, IEC 13211-1: 1995: Information Technology—Programming Languages—Prolog—Part 1: General Core, ISO: Geneva, Switzerland, 1995.

[6] J. Cheney, C. Urban, Alpha-prolog: A logic programming language with names, binding, and alpha-equivalence (2004).

[7] F. Jouault, N. Pouillard, Approaching model edition as a linear logic problem, in: 2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), 2023. To appear.