Modeling Languages
System Prototype and Verification Using Metamodel-Based Transformations
Luis Pedro, Levi Lucio, and Didier Buchs • University of Geneva

Mapping domain-specific languages' core concepts into the Concurrent Object-Oriented Petri Nets formal specification language provides users with the semantics necessary for developing prototypes for these DSLs.

Different knowledge domains demand different types of support from software languages. Domain engineers often use domain-specific languages to overcome this problem. DSLs are less comprehensive than general-purpose languages (such as C++ or Java) but much more expressive in their domain, letting domain experts understand, validate, modify, and often develop DSL programs themselves.¹

This doesn't imply, however, that you can easily and rapidly generate prototypes. DSLs strive to provide the right set of predefined abstractions for their problem domains. But they often don't include strong abstraction mechanisms (for example, constructions to allow modularity and reusability). In addition, DSLs are difficult to design, implement, and maintain and are often less efficient than hand-coded software.

To address some of these problems, we propose transforming a DSL into the Concurrent Object-Oriented Petri Nets formalism.² The DSL metamodel serves as the transformation's starting point. The transformation represents the semantic mapping between the DSL and CO-OPN. We aim both to provide a formally defined semantics for the DSL and, because we integrate CO-OPN in a framework, to provide the functionalities that allow model verification and fast prototype generation for the DSL.

Background
Our methodology involves three main concepts: the CO-OPN formalism, DSL prototyping, and model transformation.

CO-OPN at a glance
CO-OPN is a formal specification language for expressing models of complex concurrent systems. Its operational semantics make it suitable for prototyping and simulation. A coordination layer gives the designer

- an abstract way to represent interactions between modeling entities and
- an abstract mapping to distributed computations.

The CO-OPN object-oriented modeling language is based on abstract algebraic data types (ADTs) and Petri nets.³ It provides a syntax and semantics that let developers use heterogeneous object-oriented concepts for system specification. The specifications are collections of ADTs, classes, and contexts—the CO-OPN modules.
CO-OPN is suitable as a target format for DSL transformations because

- It's a modular language allowing specification of different DSL components and their relationships.
- The specifications are described in a completely abstract axiomatized fashion.
- You can fully define and explore the system states.

In addition, the CoopnBuilder integrated development environment (http://smv.unige.ch/tiki-index.php?page=IntroCoopn), permits, under some restrictions, automatic source-code generation for fast prototyping, execution and simulation of the generated prototype, enrichment of the prototype with specific code, and prototype execution and verification of implementations through testing.

**Domain-specific-language prototyping**

Our approach involves prototyping DSLs by defining a transformation into the CO-OPN formalism. So, a DSL doesn't need to provide functionalities for prototype generation and simulation. Furthermore, our approach allows easy maintenance of the DSL because it gains these functionalities after transformation, exploiting CO-OPN's semantics and related tools.

Figure 1 illustrates the four steps for generating a DSL prototype:

1. Define a metamodel in an instance of a metametamodeling formalism (the dashed box in figure 1).
2. Define rules for transformation to CO-OPN.
3. Generate the CO-OPN specification.
4. Generate and simulate the prototype.

![Figure 1. Prototyping a domain-specific language. The numbers indicate the process steps.](image-url)
Step 1 is entirely manual and in principle must be done only once. Step 2 is a dual step in that it occurs at both the semantic and syntactic levels.

We map the DSL's semantics into CO-OPN as an evaluation function. This function's domain is a program in the given DSL and the program's state. The codomain is another program state representing the evaluation's results. Evaluation semantics are a general resource for simply and formally defining the semantics of a wide range of computer languages. We've tried to abstract functional semantic constructions from various programming paradigms and to define a DSL's evaluation function as a composition of some (or all) of those semantic constructions.

At the syntactic level, we define translation rules that map the DSL syntax into the CO-OPN one based on both metamodels.

Steps 3 and 4 are fully automatic once the first two are complete.

**Metamodeling and model transformation**

The process of defining the metamodel (also known as a language's abstract syntax) and the transformation rules is crucial for our methodology. We use the Meta-Object Facility (MOF) or Eclipse Modeling Framework (EMF) Ecore formalism to define the metamodel. We also use one of these formalisms as an interface to several transformation languages—for example, the Active Template Library (ATL), Matrix Template Library (MTL) (http://modelware.inria.fr/rubrique4.html), or Mod-Transf (http://modelware.inria.fr/rubrique15.html). The Ecore formalism is expressed using a restricted form of the UML class diagram and can serve as a standardized representation of a DSL's abstract syntax.

Figure 2 shows the result of defining a metamodel using the Ecore formalism. The figure presents a simplified version of SQL's metamodel. Throughout the rest of the article, we focus on SQL as the DSL to exemplify our methodology's application. Although SQL doesn't have concurrency constructs, CO-OPN can easily manage concurrency aspects. Examples of concurrency handling and its transformation to CO-OPN are available elsewhere.
Having defined the metamodels, we must be able to generate APIs that let us express programmatically the DSL's abstract syntax. We use Java interfaces—such as the EMF model to Java mapping or Java metadata interfaces (JMI)—that are generated directly from an Ecore or MOF specification. These interfaces let us easily manipulate the model to ensure syntax-level correctness.

The simplified SQL metamodel includes

- query expressions represented by the `QueryExpression` class, which is specialized by the different types of SQL query types (CREATE, SELECT, INSERT, UPDATE, and DELETE), also represented as classes in the metamodel;
- the SQL simplified structure, represented by the `Table`, `Column`, and `SQLDataType` classes;
- the search expressions used in the `Where` clauses, produced by associating a `LeftCondition` with a `RightCondition` using operators; and
- classes representing the concepts of `from`, `into`, and `values`.

**Methodology principles**

Here are the general guidelines for our DSL prototyping methodology.

**Meeting CO-OPN concepts**

To derive a CO-OPN model from a DSL, we must be able to map the DSL concepts and semantics into CO-OPN. CO-OPN offers a high level of expressiveness. To better understand a DSL's semantics to
ensure a correct mapping, we isolated some transformation patterns, which we present later in this article.

Despite our approach's generality, we also wish to tackle concepts that are orthogonal to certain DSL classes. So, we specialize our transformation approach according to some of these concepts—for example, these:

- We map the notions of highly imperative languages (that is, sequence, loop, and selection) into CO-OPN through ADTs and evaluation functions.
- We assume that recursion mechanisms, such as those provided by the more functional or logical languages, are transformed into CO-OPN ADTs. We map the recursion feature into derived induction rules that are transformed into recursive functions.
- As a Petri net-based language, CO-OPN can cope with DSLs that handle concurrency. CO-OPN handles concurrency naturally by using multiple tokens, where they exist, or by creating independent transitions.
- We transform DSLs providing object-oriented resources as features to CO-OPN using its inheritance, class (and object), and polymorphism concepts.
- CO-OPN provides coordination modules (known as contexts) to serve as an interface, letting unrelated entities interact with a specification.

The differences between some of the language concepts lead to a transformation's tendency to follow certain rules. For example, if a DSL provides many imperative and pure-logic language features, transformations using ADTs as the main artifact will be easier. On the other hand, if the DSL involves a higher abstraction level, transformations using CO-OPN concepts will be easier. In general, the choice between ADT or class transformation depends on whether you're using components to modify states or components with concurrent behavior.

**Transformation patterns**

Our goal is to isolate the maximum number of possible patterns to help users define their transformations. This also achieves a more efficient language transformation.

Here are some of the isolated patterns:

- **Data types** are transformed directly to ADTs using existing CO-OPN foundation classes (CFCs)—predefined libraries defining a set of ADTs for the most commonly used data types.
- **Boolean expressions** are expression evaluations that return a Boolean value. You can isolate this pattern from an occurrence such as `bool_expr = a bool_operator b`, where `a` and `b` can be variables or another Boolean expression. This kind of pattern should be transformed into axioms of the CO-OPN ADTs.
- **Loops** typically are a language's iterative statements. Loops are usually denoted by `while`, `do`, or `for` and should be transformed into evaluation functions of CO-OPN ADTs.
- **Selection** represents the behavior of a classic `if a then b else c` expression. Such a pattern's occurrence should be transformed into ADT evaluation functions or axioms of CO-OPN classes, depending on whether the language is imperative.
- **Inclusion** is similar to Java's `import` or C/C++'s `include`. In CO-OPN, this pattern will be transformed into a similar concept represented by the `use` keyword.
- **Class** describes a collection of encapsulated instance variables and methods (defining the class's behavior). Languages that provide object-oriented features typically include this notion (with some syntax), which is directly mapped into CO-OPN classes using the associated methods. The class logic must be transformed into positive conditional equational axioms.
- **Object** is an instance of a class and included in languages with object-oriented concepts. Objects are transformed into instances of CO-OPN classes and, if necessary, stored in Petri nets places of the same type as the class.
- **Interface** lets you define the possible interactions that others might have with the specification.
To incorporate our methodology into a framework, we provide the functionalities with which you can express these (and other) patterns in terms of their occurrences in the DSL metamodel. Then, you can use the predefined transformation definitions to associate patterns with their occurrence in the metamodel.

The same pattern can appear in completely different languages and with a completely different syntax. For example, consider the `Where` clause in the SQL metamodel and the `selection` statement in the C metamodel. Both of these occurrences represent the same behavior, resulting in a transformation that should be similar despite syntax differences.

**Application example**

We use the SQL metamodel and semantics to illustrate our approach. For languages such as SQL, the best transformation approach is to mix CO-OPN classes and ADTs, because SQL uses concepts of both of structure (the tables) and more abstract concepts of behavior such as inserting or selecting table records. The transformation from SQL to CO-OPN follows these rules:

- For the SQL data structure, we map the transformation to CO-OPN ADTs—that is, we map each $n$-tuple identifying the table’s columns to a specific ADT when we perform a `CREATE table` query.
- We map each SQL table to a CO-OPN class with two methods—`INSERT` and `SELECT`—that act as operations to be performed on the class.
- The tables serve as the underlying Petri net places for the CO-OPN class.
- The Petri net places are an $n$-tuple mapping the number of columns and their types.

Figure 3, an instance diagram of the SQL metamodel in figure 2, illustrates this procedure. The diagram represents the instantiated classes needed to define an SQL `INSERT` query and is a hierarchical view of a query of the type `INSERT INTO aTable(name, age) Values (joe, 28)`.

![Figure 3. An INSERT query expression.](image-url)
This example also includes a simple database structure composed by a table `aTable` with two columns: `(name = name)` and `(name = age)`.

We map this structure as a CO-OPN ADT with an $n$-tuple record addressing the equivalent structure (see figure 4). The ADT defines the static data structure transformed by creating a new generator according to the table structure, by mapping each column type to its equivalent in the CFCs. For example, the `Integer` type is transformed into the `Natural` CO-OPN ADT, and the `VarChar` type is transformed into the `String` ADT. The ADT also defines the axioms that are generated to cope with the operational behavior on data types. We deduce and implement the same operations defined in the ADTs of the table columns for the ADT corresponding to the $n$-tuple. These operations are expressed in figure 4 by the `Operation` field.

```
ADT ADTaTableRecord;
Interface
Use
    Booleans;
    Naturals;
    String;
Sort
    atablerecord;
Generator
    newaTableRecord _ _ : string natural -> atablerecord;
Operation
    _ = _ : atablerecord atablerecord -> boolean;
Body
    Axiom
    (newaTableRecord n m = newaTableRecord n1 m1) =
    ((n = n1) and (m = m1));
    Where
    m : natural;
    n : string;
    m1 : natural;
    n1 : string;
End ADTaTableRecord;
```

**Figure 4.** The `ADTaTableRecord` ADT corresponds to the SQL table structure.

After transforming the table structure, another mechanism adds a new place to the CO-OPN `Database` class. This place has the same name as the place in the metamodel instance (`aTable` in this case), and its type is the one defined by the corresponding ADT (`ADTaTableRecord` in this case).

Assuming that a `Database` class exists for each database, the transformation creates a Petri net place `myTable` of the type `aTableRecord`. Figure 5 illustrates this process. The general approach is to transform `INSERT` constructions, such as the one in the example, into CO-OPN by calling the method `INSERT` with a parameter `newaTableRecord`.

The `Database` class will have simple axioms for `INSERT` purposes that add a new token in the Petri net place `aTable` (initially empty). Figure 5a shows a simplified view of the `Database` class before the `INSERT` query executes. Figure 5b shows that class after the query execution. It illustrates the result of calling the method `INSERT INTO _ < _ , _ >`, where the first underscore is for the table name and the last two correspond to the values to be inserted. For each table in the database, the transformation generates one `INSERT` method according to its structure.
Figure 5. Mapping an SQL table into a CO-OPN class: the Database class (a) before the INSERT query executes and (b) after the query executes.

As figure 6 shows, in the Database class, the query generates the method SELECT _ FROM _ SQLWhere _ _ _ : string string string char string. The first underscore is a string that specifies the attribute field to be selected; the second represents the table name. The last three are the column name on which to perform the WHERE operation, the operator name, and the value to compare with.

```
Class Database
Interface
Use
   ADTTableRecord;
Type
database;
Gates
   % ON Database % _ : string natural;
Methods
   INSERT INTO Database _ : atablerecord; getNumRecords;
   SELECT _ FROM _ SQLWhere _ _ _ :
      string string string char string;
Body
Places
   aTable _ : atablerecord;
Initial
   numRecords 0;
Axioms
   INSERT INTO Database r:: -> aTable r;
   (comparator = eq) and (tableName = t) and (field = age)
       and (fieldSearch = name) =>
   SELECT field FROM tableName
   SQLWhere fieldSearch comparator value With
       this . % field ON Database % variableAgeName::
   aTable newaTableRecord value variableAgeName ->
   aTable newaTableRecord value variableAgeName;
Where
   r : atablerecord;
   n : natural;
   comparator : char;
   value : string;
   variableAgeName : natural;
   tableName : char;
End Database;
```

Figure 6. An axiom and methods for inserting tokens into places in the CO-OPN classes.
This lets us execute queries of the type `SELECT age FROM aTable Where name='joe'` by calling the method previously defined as `SELECT age FROM aTable SQLWhere name eq joe`. This automatically produces an output result of the type `% age ON Database % 28.

We obtain this result because the axiom in figure 6 specifies the behavior of an SQL `SELECT` query. The figure shows only one axiom, but we generate variations of it to cope with all cases—such as other operators or queries in other fields.

We can easily transform constructions such as loops and if-then-else using ADT evaluation functions or positive conditional conjunctive axioms. We present the approach's generality and describe other constructions elsewhere.\textsuperscript{12}

The transformation process and composition

Having isolated some patterns for transforming DSL semantics into CO-OPN semantics, we should be able to automatically compose them. To understand how we can do this, consider the transformation process and execution as a combination of smaller transformations—each representing a transformation pattern.

We're working on case studies through which we can define the transformation composition and, consequently, transformation execution sequences.

Prototype generation and execution

We transform CO-OPN specifications into a Java package including ADT classes and JavaBean-like components. To use these components, we create a `CoopnTransaction` object to call their methods. We capture a CO-OPN model's output events using an `EventListener`—that is, adding `EventListener` to the corresponding `Gate`. Stanislav Chachkov describes the technical details of generating Java code from CO-OPN specifications.\textsuperscript{13} Other work describes how to use the generated code in an application and interface with existing Java libraries.\textsuperscript{14,15}

CoopnBuilder also provides a prototype simulation environment via Java reflection functions. It lets users interpret the prototype as it's integrated into an application. A graph shows the evolution of CO-OPN model states, with each node representing a unique state. Each time a method is called and the system state changes, an arc appears between the old and new states. A new node is created if the new state hadn't previously appeared during the simulation. This graphical simulation lets users manually explore the system's state space. For systems with large state spaces, users can dump the complete simulation trace into an XML file and use automatic analysis tools to deduce properties such as the coverage of state space during the simulation.

As we started working on our methodology, we began developing tools to support it. We've already deployed some of the features presented, such as the metamodel and model generic browsing (tree-based), automatic generation of Java interfaces for a metamodel, and model exploration. We've also developed functionalities such as UML-like generic metamodel browsing. Tools that allow for specific model browsing, editing with graphical support, and specification are under development. Meanwhile, we're trying to isolate the biggest possible number of patterns, build a catalog of how to transform them, and define how to easily use and compose them. We're also working on defining the transformation composition behavior and result.

Integration of the transformation languages mentioned in this article is the next and natural step. We aim to integrate them and generate tools that can cope with the complexity of model transformation techniques yet remain intuitive enough to be usable.
References


Luis Pedro is part of the Software Modeling and Verification group of the University of Geneva’s Computing Department. He’s also working on his PhD thesis on domain-specific-language prototyping and verification, modeling, and metamodeling transformation and composition. His research interests include software modeling and verification, fast prototyping of domain-specific languages, and (meta)model composition for semantic enrichment of syntactic domain concepts. He has a degree in physics engineering from the University of Lisbon with a specialization in database technology for control and real-time systems. Contact him at the Software Modeling and Verification Group. Univ. of Geneva, CUI, 24 Rue du Général-Dufour, CH-1204 Genève, Switzerland; luis.pedro@cui.unige.ch.
Levi Lucio is writing his PhD dissertation on model-based test case generation from the Concurrent Object-Oriented Petri Nets specifications, at the University of Geneva's Software Modeling and Verification group. His research interests revolve around modeling and verification using formal methods. He’s particularly interested in formalizing and studying the properties of the semantics of programming and specification languages. He obtained his MSc in database technology and software engineering from the University of Sunderland. Contact him at the Software Modeling and Verification Group, Univ. of Geneva, CUI, 24 Rue du Général-Dufour, CH-1211 Genève 4, Switzerland; levi.lucio@cui.unige.ch.

Didier Buchs is an associate professor of the faculty of science at the University of Geneva's Centre Universitaire d'informatique. His main research interests are formal specification methods and testing techniques for real-size distributed systems. He received his PhD in computer science from the University of Geneva. Contact him at the Software Modeling and Verification Group, Univ. of Geneva, CUI, 24 rue du Général-Dufour, CH-1211 Genève 4, Switzerland; didier.buchs@cui.unige.ch; http://smv.unige.ch/tiki-index.php.

Related Links
- DS Online's Software Engineering Community (http://dsonline.computer.org/portal/site/dsonline/index.jsp?pageID=dso_level1&path=dsonline/topics/software_engineering&file=index.xml&xsl=article.xsl)

Cite this article: