Separation of Concerns in Model-Driven Development

Vinay Kulkarni and Sreedhar Reddy, *Tata Research Development and Design Centre*

Model-driven development has improved productivity, quality, and platform independence, but it hasn't been that successful in supporting reuse and system evolution. The proposed Template abstraction addresses this problem in an integrated way by dealing with separation of concerns at both the model and the code level.

To facilitate traceability, reuse, and evolution, systems should be specified as compositions of clearly separated and separately specified concerns of interest. Customizing model-driven development environments—for example, to support different application design strategies—is difficult because they lack support for clear separation of concerns. Moreover, because we typically specify MDD systems in terms of models and code, we must address the issue of separation of concerns at both the model and code levels.[1] The Template abstraction we propose lets us model applications as hierarchical compositions of templates. We illustrate our composition process using a unified metamodel.

Model-driven development

MDD aims to shift the focus of software development activity from coding to modeling. Application development starts with an abstract specification A, which is to be transformed into a concrete implementation C on a target architecture, as Figure 1a shows.[2] The target architecture is usually layered, with each layer representing one view of the system.

The modeling approach constructs A using different abstract views $A_1 \ldots A_n$, each defining a set of properties corresponding to the concern it models. A view, A_i, is an instance of a more general structure that we can represent as a (meta)model M_i, such as the user interaction model for sequences of user-system interactions, or an entity-relationship model for representing data. A is usually not available separately: We use it here to represent the composition of the views $A_1 \ldots A_n$. We can transform each A_i into C_i, with application-level composition of $C_1 \ldots C_n$ giving C the intended implementation of A. Instead of performing such a transformation for every

0740-7459/03/$17.00 © 2003 IEEE

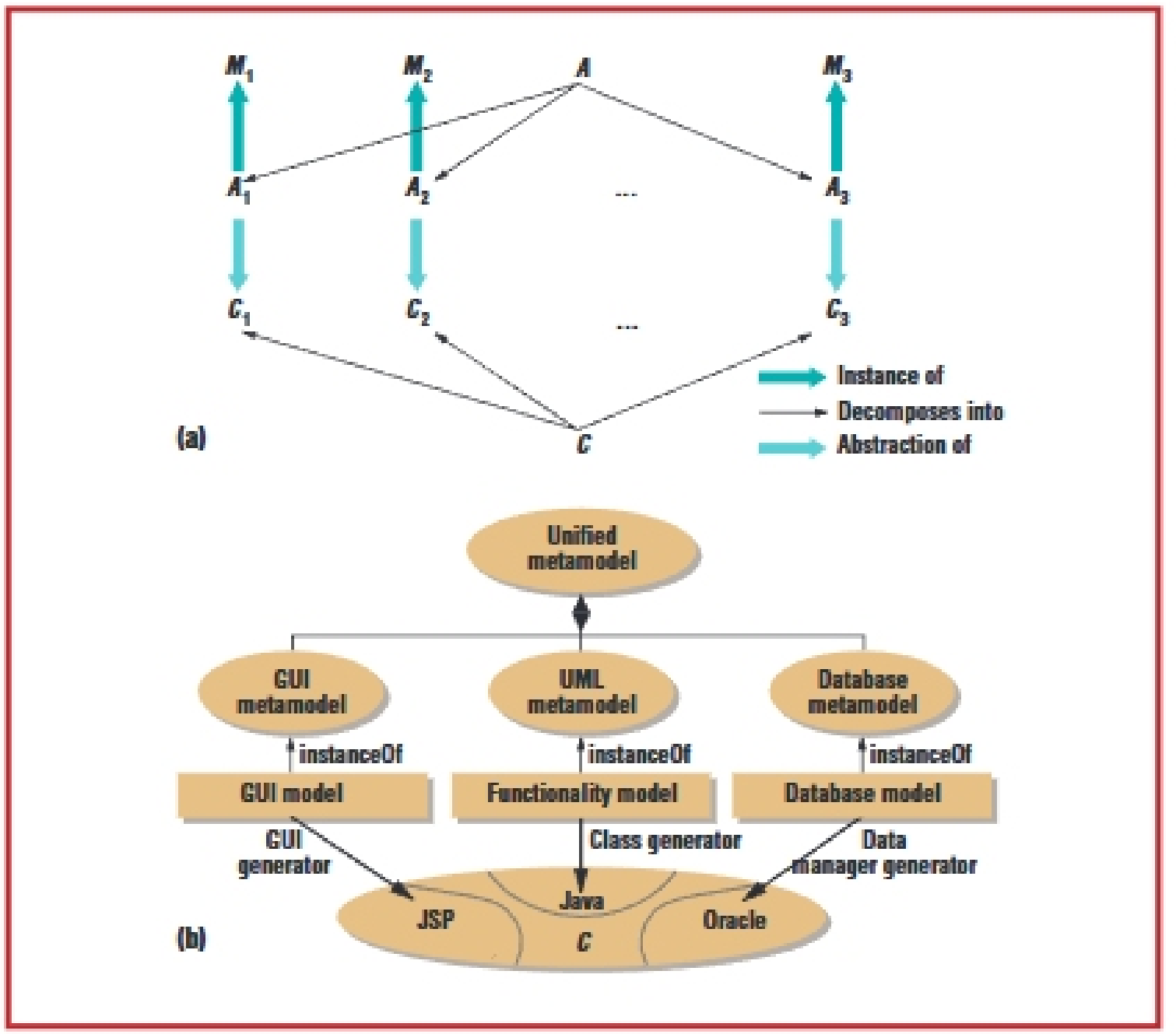

Figure 1. (a) Model-driven software development; (b) layered architecture of a typical business application.

application manually, we use M_t, of which A_t is an instance, and implement generic transformations at the model level. We can apply these transformations to all instances of M_t. Defining transformations at the level of M_t rather than A_t makes scaling up the method to handle large programs possible. For example, we can specify a transformation from a class diagram to Java classes and then apply this transformation to generate Java classes for any application.

Figure 1b shows a typical business application modeled according to MDD. The application is implemented across three architecture layers—user interface, application functionality, and database—each on a different platform supporting different primitives. For example, user interface platforms such as Visual Basic provide windows and controls as implementation primitives. The application logic is implemented in a Java-like programming language with classes and methods as primitives, and the database layer is built in a relational database system using tables and columns. The three layers are then combined to get an implementation C.

Each layer must address several concerns, for example:

- The GUI layer must address a standard look and feel across screens, standard user interaction patterns, and the mapping of window controls to application classes.
- The functionality layer must address error handling and logging.
- The database layer must address concurrency, auditing, and locking.

Each concern should be specified in a modeling notation best suited for its expression. A UML extension might be suitable, as a profile or variant, where standard UML notations are inadequate.

Separation of concerns

A *parameterized package* is a parametric model element that captures patterns of recurring structure, behavior, and constraints of models. We create parameterized packages using the package extension mechanism.[3] When

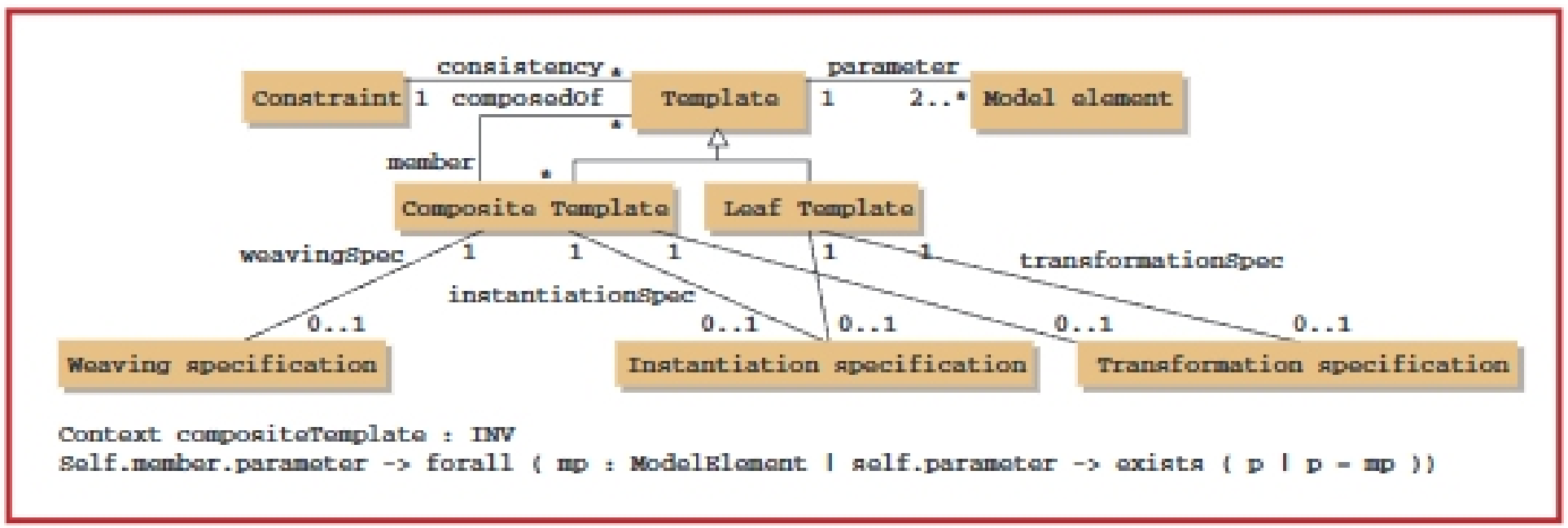

Figure 2. The Template metamodel.

model elements are supplied as parameters to the package, the package creates a set of new model elements. Our Template abstraction uses parameterized package abstraction[4] to address the separate specification and composition of model patterns, and aspect-oriented programming[5] to address separation of concerns at the code level.

Figure 2 shows the Template metamodel, which expresses how a concern specification is transformed into models and code. There are two kinds of Templates: a Leaf Template and a Composite Template. The Leaf Template's instantiation specification specifies how to create concern-specific model elements, and its transformation specification specifies how the instantiated model is transformed into platform-specific code. The Composite Template's instantiation specification specifies how model elements constructed in member Templates are merged (woven together); we have found that the merge-by-name scheme of model merging has worked well for our purposes. To specify transformations, you can use any model-aware transformation language; we used SpecL.[6] The Template's weaving specification specifies how the code generated by its member Templates is merged; a code-weaving specification language along the lines of Hyper/J[7] has been sufficient for us.

We compose applications by doing a post-order traversal of the Template hierarchy in three sequential steps:

1. *Instantiation:* stamps out models and merges them
2. *Transformation:* transforms models into code and generates weaving specifications for composing the generated code
3. *Weaving:* composes the generated code fragments by processing the weaving specifications

This process lets us specify applications as hierarchical compositions of Templates of interest.

An example

Figure 3 shows a unified metamodel, based on Figure 1b, with persistent and auditable classes together with their table representations. (We can mark up a static UML model so that classes are independently persistent and auditable.) A persistent class has a representation as a database table. An auditable class records the history of its state changes. We generate a model's persistent and auditable features from a static model consisting of classes and attributes, as defined in Figure 3. Figure 4 shows the Template model for this application.

This example considers the functionality and data manager layers shown in Figure 1b that support the concerns of the Object-Relational Map, Audit, and Object Model Translation Templates. All Templates in this example take `Class` as the input parameter. Following are the Templates' specifications.

Object-Relational Map

This Template specifies an object-relational mapping strategy for a persistent class.

Instantiation specification. For an input class Cl having its `isPersistent` property set to True, there should exist

- A table T having the same name as the class Cl
- A key K having the same name as the class Cl
- A `mapsTo` association between T and Cl