Conceptual-Model Programming: A Manifesto

David W. Embley, Stephen W. Liddle, and Oscar Pastor

1 Preamble

In order to promote Conceptual-Model Programming (CMP), we set forth these CMP Articles. We hold these articles to be the defining principles for model-complete software development.

In essence, this CMP manifesto asserts that programming activities are to be carried out via conceptual modeling. For applications amenable to conceptual-model designs, software developers should never need to write a line of traditional code. Thus, programming is actually "Conceptual-Model Programming" ("CMP").

To accommodate CMP, conceptual-modeling languages must be executable. They must also be capable of completely deploying both databases and user interfaces and conceptually expressing database access and user interaction. To enable CMP, a conceptual-model compiler must exist to generate underlying code (which could be, but is not necessarily, high-level-language code that itself needs further compilation). Important, however, is that model-compiled code is beyond the purview of CMP programmers—both for initially creating the application system being developed and for enhancing or evolving the application system. Thus, application-system development becomes entirely model-driven, and CMP constitutes model-complete software development.

David W. Embley
Brigham Young University, Provo, Utah 84602, USA e-mail: embley@cs.byu.edu

Stephen W. Liddle
Brigham Young University, Provo, Utah 84602, USA e-mail: liddle@byu.edu

Oscar Pastor
Valencia University of Technology, 46022 Valencia, Spain e-mail: opastor@dsic.upv.es

2 CMP Articles

Conceptual modeling is programming. The conceptual-model instance is the code (instead of: "the code is the model"—"the model is the code"). A conceptual-model compiler assures that program execution corresponds to the conceptual specification, thus making the conceptual-model instance directly executable.

The conceptual model, with which modelers program, must be:

- ***complete and holistic.*** The conceptual model must provide a holistic view of all application components. It must include all necessary aspects of data (structure), behavior (function), and interaction (both component interaction and user interaction).
- ***conceptual but precise.*** The conceptual modeling elements must be precisely defined and must be based on an ontological agreement that fixes the concepts and their associated notation. Parsimony should guide, but not rule, both the modeling elements and the notation.

Application evolution occurs at the level of the model. Conceptual-model programmers should evolve an application through the model instance, not through generated, lower level code.

3 Exposition

The principles of the CMP Articles are tenable only if: (1) a conceptual-model instance is executable (Section 3.1) and (2) programmers can do all their development work by specifying a conceptual-model instance for their application (Section 3.2).

3.1 Executable Conceptual Models

Conceptual-Model-Programming (CMP) is about precisely capturing an application in the language of an executable conceptual model that is sufficient for all storage, functional, and interaction requirements of an application. Precisely capturing an application as a conceptual-model instance *is* programming—i.e., is conceptual-model programming, CM programming, or CMP.

To illustrate CMP, Figures 1–8 show some sample conceptual-model specifications. These sample specifications are about a free-lance photography agency. Free-lance photographers register with the agency. They then submit annotated pictures. An evaluator for the agency determines which pictures

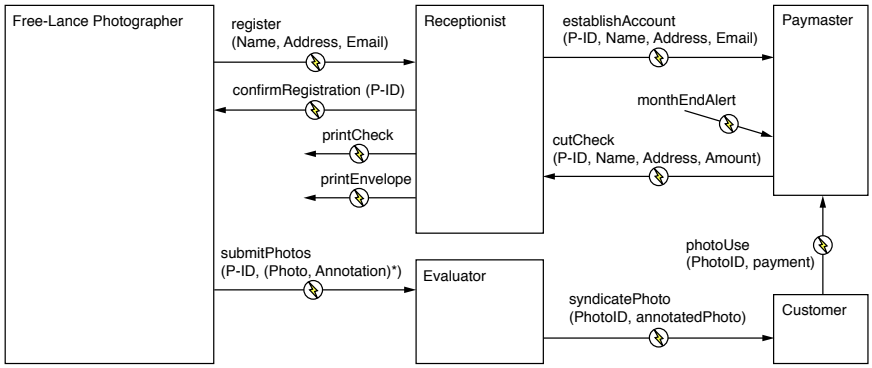

Fig. 1 Sample CMP Component Interaction Diagram.

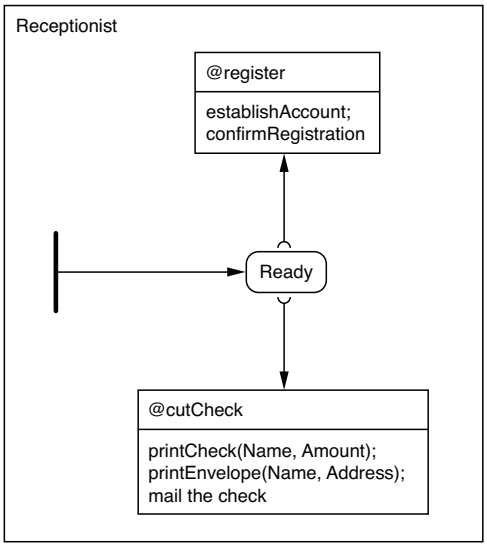

Fig. 2 Sample CMP Behavior Diagram.

to syndicate. Customers use syndicated pictures and pay royalties. The company pays free-lance photographers a percentage of the royalties and keeps the rest.

The particular notation of the conceptual-modeling language is not important, except that it is conceptual. What is important is that a collection of conceptual-model specifications provides all the information needed to generate a fully executable application.

Figures 1–4 show some generic samples covering the full range of development activities from specifying database storage structures, through stipulating behavior and component interaction, to describing user-interface data exchange. They represent a coherent collection in which cross-diagram objects

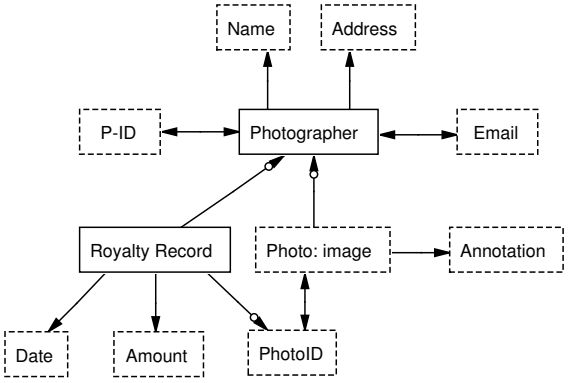

Fig. 3 Sample CMP Database Structure Diagram.

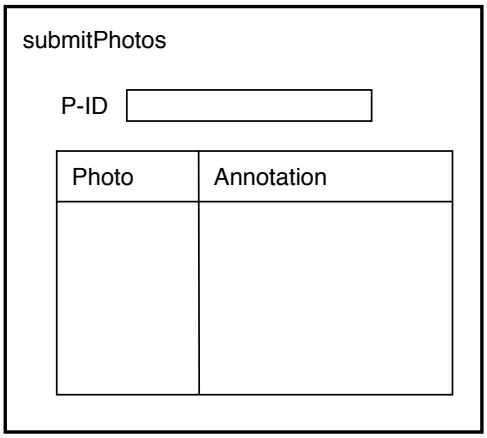

Fig. 4 Sample CMP Conceptual User Interface Specification.

and components have the same name. Together, they, along with additional diagrams needed to complete the full specification, constitute a CM program for the free-lance photography agency.

Figures 5–8 illustrate alternative graphical notation and also serve to indicate that the collection of conceptual-model diagrams constituting a CM program need not all be of the same genre. Figure 5 is a UML communication diagram that corresponds to the interaction diagram in Figure 1. Figure 6 is a Statechart that corresponds to the behavior diagram in Figure 2. Figure 7 is an Entity-Relationship (ER) diagram that is semantically equivalent to the structure diagram in Figure 3. And, Figure 8 is an Olivanova user-interface specification that not only establishes the data to be exchanged, as expressed in Figure 4, but also establishes the appearance of the interface a

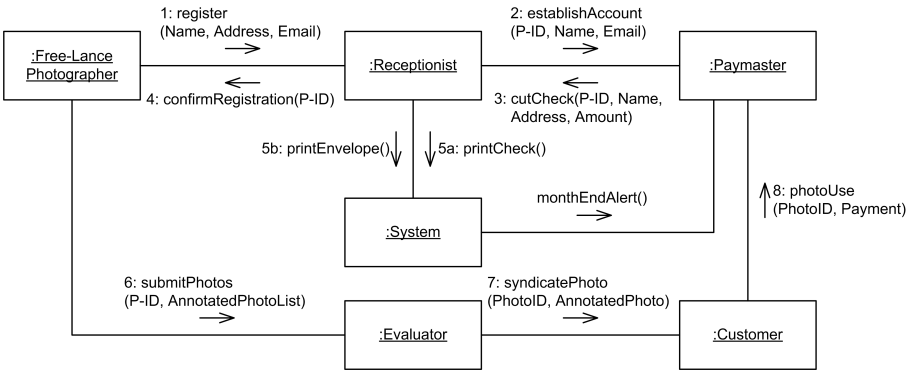

Fig. 5 UML Communication Diagram Equivalent to the Component Interaction Diagram in Figure 1.

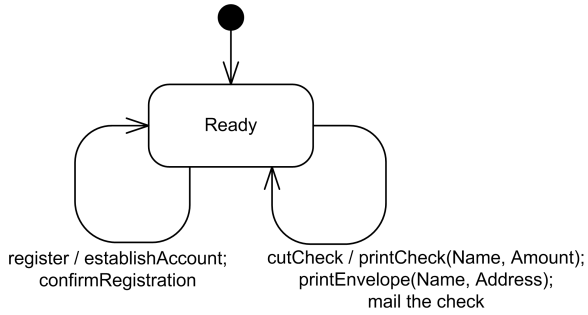

Fig. 6 Statechart Diagram Equivalent to the Behavior Diagram in Figure 2.

user of the free-lance photography sees when submitting photos for potential syndication.

Observe that in all diagrams fundamental constructs have two-dimensional, graphical representations. Behavior diagrams express control flow graphically; interaction diagrams express sending and receiving actions graphically; database structure diagrams express entities, relationships, and constraints graphically; and user interaction diagrams express data exchange and the look-and-feel of a user interface graphically. Text associated with graphical constructs provides names for objects and components, expressions that naturally appear as text, and connecting syntax.

Although the ability to render fundamental conceptualizations graphically is a requirement, actually rendering them graphically is not. CM programmers may express conceptualizations in purely textual languages, so long as the languages are "*model-equivalent*." In a model-equivalent language each fundamental construct has an isomorphic correspondence to a graphical representation. Figure 9 shows some examples. *Photographer [1] Name [1:*]* in Figure 9 corresponds to the functional edge between the nodes *Photographer*

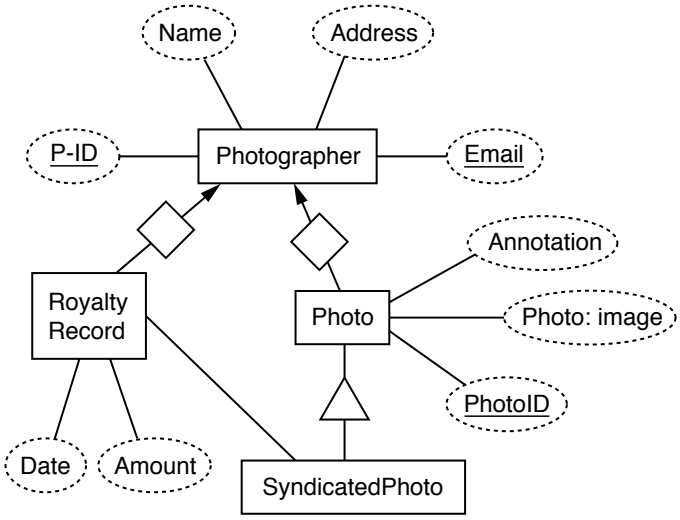

Fig. 7 Entity-Relationship Diagram Equivalent to the Database Structure Diagram in Figure 3.

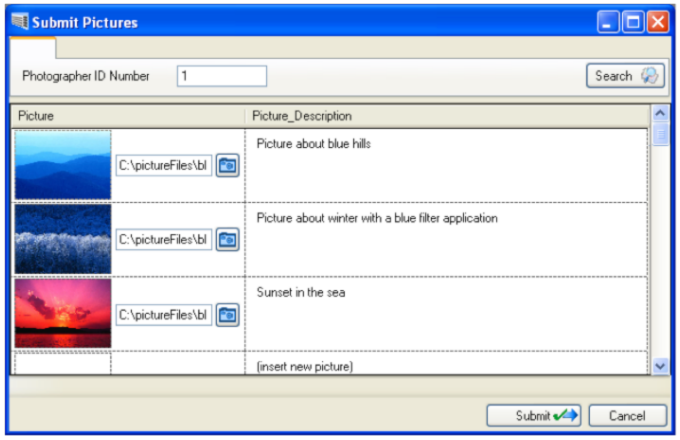

Fig. 8 Olivanova User Interface Specification Equivalent to the User Interface Specification in Figure 4. (Note: An additional conceptual specification exists that associates the external names "Submit Pictures", "Photographer ID Number", "Picture", and "Picture_Description" respectively with the internal names "submitPhotos", "P-ID", "Photo", and "Annotation". Also, an additional top-level conceptual specification exists to allow a photographer to navigate to this "Submit Pictures" interface.)

and *Name* in the database-structure graph in Figure 3. The *[1]* and the *[1:*]* are participation constraints; thus, each *Photographer* object associates with exactly one *Name* object, making the relationship functional. In Figure 2, the circled *Ready* denotes the potential for an object to be in the ready state—*when Ready* in Figure 9 denotes the same; both the arrows whose tails are

```
...
Photographer [1] P-ID [1];
Photographer [1] Name [1:*];
...
@initialize Receptionist
enter Ready
end;

when Ready new thread
@register(Name, Address, Email) then
...
end;
...
```

Fig. 9 Model-Equivalent Textual Representation.

disconnected from the *Ready* state in Figure 2 and *new thread* in Figure 9 denote spawning new threads of control; and the Event-Condition-Action (ECA) box with the event *@register* in Figure 2 matches through its name with the interaction *register(Name, Address, Email)* in Figure 1. Allowing experienced CM programmers to express conceptual-model instances textually provides for economy of expression without loss of conceptualization. Ideally, CM programmers and analysts can be at either extreme (no graphics / all graphics) or at a comfortable place in between.

To see that conceptual-model instances can be fully executable, consider the diagrams in Figures 1–3. In interaction diagrams such as the diagram in Figure 1, message passing is executable if in the code the point of initiation of the message is known, the information to be passed is known, and the point of reception of the message is known. An interaction such as *establishAccount(P-ID, Name, Address, Email)* in Figure 1 specifies the information to be passed and provides a name for reference within specified origin and destination active objects. The tail side of the interaction arrow specifies the origin (*Receptionist* for *establishAccount*), and the head side specifies the destination (*Paymaster* for *establishAccount*). Within the behavior diagram of active objects, an appropriate reference to the name specifies the point of initiation in the originating behavior diagram and the point of reception in the receiving behavior diagram. In the behavior diagram in Figure 2, for example, *establishAccount* in the ECA box initiates the interaction *establishAccount(P-ID, Name, Address, Email)* in Figure 1, and *@register* is the point of reception for the interaction *register(Name, Address, Email)*, also in Figure 1.

Behavior diagrams require a full specification of the control flow. The behavior diagram in Figure 2, for example, consists fundamentally of a collection of ECA rules: when events (marked by @) occur, if an object's thread is in a prior state and specified conditions (if any) hold, the ECA rule fires. Thus, for example, when a thread of control is in the *Ready* state in Figure 2 and a *Receptionist* receives an *@register* message, the ECA rule fires, spawning a thread of control to establish an account and confirm the registration. The

new thread of control then dies, but the original thread of control remains active in the *Ready* state. In addition to full specification of control flow, the events, conditions, and sequence of statements in ECA rules must be formal enough to be compilable into code. In Figure 2, the *@register* ECA rule is fully formal: both the event and the actions reference fully specified messages in the interaction diagram in Figure 1. The *@cutCheck* ECA rule is also fully formal if the actions are all primitive or provided in a library. Alternatively, if the *Receptionist* is actually a human user of the system, all the ECA rules are sufficient as instructions. Further, each fully specified message implicitly has a corresponding interface form (e.g., Figure 4 for the *submitPhotos* message in Figure 2), which can be directly implemented (as-is) or visually enhanced to be more pleasing with an improved user-interface specification (e.g., Figure 8).

Structure diagrams must fully specify the database schema. From the conceptual-model instance either in Figure 3 or in Figure 7 the CM compiler can infer the SQL schema in Figure 10. From Figure 7, for example, the mapping algorithm generates each entity as a table with its associated attributes and foreign-key references. Then, since the attribute *Photo:image* for the entity *Photo* is an image, which is to be implemented with the type *BLOB*, the mapping algorithm generates the attribute *Photo:image* as a weak entity and thus as the table *PhotoFile*, which is dependent on the table *Photo*. Additional constraints, such as check constraints and alternative-type constraints, can be added to the conceptual database structure diagram and propagated into a formal schema specification. The type specification *image* in Figure 3 is an example; specifying *Amount:smallmoney* in place of *Amount* in Figure 3 would be another example.

3.2 Conceptual Modeling and CMP

CMP development work includes analysis, specification, design, implementation, deployment, enhancement, and evolution. CM programmers work through every stage conceptually, writing all descriptions in a conceptual-modeling language. Typically, initial stages are informal—progressing through the stages is a process of formalizing the CM descriptions until in the implementation stage they are fully formal and ready for deployment. Subsequent enhancement and evolution makes direct use of CM descriptions, which are kept for this purpose. CM programmers should never discard deployed conceptual-model instances (the executable conceptual-model instances are the code), and CM programmers should neither enhance nor evolve deployed applications by altering compiled code, but rather always by altering and recompiling conceptual-model instances.

The notion of "tunable formalism" plays an interesting role in CMP. The idea is that formalism in conceptual-model descriptions can be "tuned"

```
CREATE TABLE Photographer (
    P-ID VARCHAR(30) PRIMARY KEY,
    Email VARCHAR(30) NOT NULL UNIQUE,
    Name VARCHAR(30) NOT NULL,
    Address VARCHAR(30) NOT NULL
    );

CREATE TABLE Photo (
    PhotoID VARCHAR(30) PRIMARY KEY,
    Annotation VARCHAR(30) NOT NULL,
    P-ID VARCHAR(30) NOT NULL REFERENCES Photographer
    );

CREATE TABLE PhotoFile (
    PhotoID VARCHAR(30) PRIMARY KEY REFERENCES Photo,
    Photo BLOB
    );

CREATE TABLE RoyaltyRecord (
    RoyaltyRecordID INT PRIMARY KEY,
    Date DATE NOT NULL,
    Amount VARCHAR(30) NOT NULL,
    PhotoID VARCHAR(30) NOT NULL REFERENCES Photo,
    P-ID VARCHAR(30) NOT NULL REFERENCES Photographer
    );
```

Fig. 10 Generated Database Schema.

"down" or "up" depending on the needs of the development team. When tuned down, clients, who contract with software-development teams to produce application software and who are typically untrained in CMP, can usually read and understand informal conceptual-model descriptions. Thus, CM analysts can directly use conceptual-model descriptions, whose formalism is tuned down, to enhance communication between clients and CM programmers. When tuned up all the way, the application is fully implemented. In between, CM programmers can read and understand the developing application abstractly and can begin to see various parts of the system execute as they become formal enough for emulation or compilation.

CMP accommodates various development strategies. CM developers need not complete one stage of the process before moving on to the next, and various parts of the application can be at different development stages at the same time. CM developers can forge ahead with the development of a kernal for the application and then treat the remaining development as enhancement and evolution.

At each stage of development CMP offers abundant opportunities for managing software development and for enhancing communication among development team members and between team members and clients. We offer a few insights:

- *Analysis* is about understanding an application and documenting that understanding. A strength of conceptual modeling is its ability to promote a common understanding within a heterogeneous development team. Conceptual modeling serves analysts well in their role of a "go-between"—it facilitates precise and concise communication with both clients and programmers. Clients can understand abstract conceptual models with the formalism tuned down; programmers tune up the formalism to make the application executable. Clients, analysts, and programmers all use the same CM notation, which results in better communication.
- *Specification* is about producing a detailed and precise proposal for a system. A difficulty with specification is that clients often do not really know what they want until they see it. Prototyping helps alleviate this concern, and CMP facilitates prototyping. Conceptual models with tuned up formalism can execute fully, but even with the formalism tuned down, they are still executable. Every CM diagram is executable as a prototype. When an emulator encounters an informal statement, it can explain its state and display the informal statement it is "executing," and it can accept user input to allow it to continue to operate and show in mock-up style how the application works. Mock-ups of end-user interfaces can be real since their specification automatically allows them to execute as part of the CM application. One view of CM programming is that it is about quickly developing a prototype and then enhancing the prototype until it becomes the deployed application.
- *Design* is about organizing a software system to achieve its goals—e.g., efficiency, maintainability, extensibility, and similar properties. An example of design is database normalization; a CM designer can use standard conceptual-modeling techniques to canonicalize a structure-model diagram to guarantee that a CM compiler's database-schema generator produces a normalized schema. A CM compiler should, as a matter of course, optimize the code it generates, but when optimization depends on "proper" conceptualization, as it does for database normalization, the CM designer should organize conceptual-model instances so that the CM compiler generates optimal code. CMP naturally promotes maintainability and extensibility. Conceptual-model diagrams are the high-level code. Because CMP compiles models into executable systems, the models cannot, as so often is the case with conceptual diagrams, be either summarily discarded or left in a disheveled state not synchronized with nor updated to match the deployed application.
- *Implementation* is about faithfully translating a design into code. For CMP, this translation is automatic. Thus, implementation requires zero effort. This does not mean, however, that application development is effortless. Rather, it means that the effort is shifted upstream. The emphasis is on analysis and specification, rather than on translating designs to programming-language syntax. Significantly, software created via CMP is "defect free" with respect to the implementation layer. If the model com-

piler faithfully translates higher level specifications into lower level code, then the only defects that can occur in a CMP-generated system are either design, specification, or analysis issues or problems with standard libraries. Thus, by avoiding implementation-layer defects, CMP promotes early detection of design-level defects.

- *Deployment* is about delivering the application system for client use. Because CM programs are immediately executable, at least in prototype fashion, pre-alpha, alpha, and beta releases follow naturally as CM programmers proceed through analysis and specification. Eventual deployment is a natural consequence of fully formalizing and properly organizing conceptual-model instances in accordance with client requirements.
- *Enhancement* and *evolution* are about making deployed applications better serve end users. In one sense, enhancing and evolving a deployed CMP application is no different from enhancing and evolving an application coded in a high-level language, except that CM programmers continue to work at a conceptual level rather than at the syntax level of the high-level language. Often, however, when evolving code written in a high-level language, enhancement and evolution require re-conceptualizing some parts of the application to serve as a starting place for improvements—either through reverse engineering or by updating and synchronizing conceptual-model instances. Although this step is often both necessary and costly when programming in a high-level language, it is never necessary and never costs anything in CMP application development because the code is already the model, which renders re-conceptualization unnecessary.

Appendage

1. Principles similar to CMP expounded by others:

   Others have set forth principles similar to CMP. In 2004, Brown, Iyengar, Rumbaugh, and Selic published *The MDA Manifesto* expounding the principles of Model-Driven Architecture [BBI+04]. The MDA Manifesto has three tenets:

   1. **Direct representation**: reduce distance between problem domain and software representation;
   2. **Automation**: mechanization of facets of software development that do not depend on human ingenuity and, especially, mechanization of bridging the gap between problem-domain representation and software representation; and
   3. **Open standards**: open-source development and accepted industry standards.

   The CMP Manifesto harmonizes well with the MDA Manifesto. The CMP Manifesto, however, takes automation a step further. It insists that conceptualizations are to be fully executable so that there is no gap between a conceptualization and a software representation. A CMP conceptualization is a software representation. Although not opposed to domain-specific modeling languages, generic, all-purpose

conceptual-modeling languages must be among the languages available for application development. Ideally, CM programmers should have a variety of notational choices. Domain-specific notation is acceptable, and perhaps even preferable, but to be a CMP conceptualization, a domain-specific conceptualization must be executable.

2. Cautions about CMP:

In 2008, Selic wrote *MDA Manifestations* [Sel08], a commentary on the MDA Manifesto. Selic's commentary includes cautions about Model-Driven Development (MDD). He asserts that MDD likely requires:

1. education (shift in view to understanding clients and users and especially an increase in the introduction of MDD methods in software-engineering education);
2. a comprehensive and systematic theory of MDD (modeling language semantics and design, model transformations, model analysis of safety and liveness properties, model-based verification, model management, MDD methods and processes, and tools); and
3. standards (the key to success of any widely used technology).

Selic believes that the shift to MDD is likely to be gradual. He also believes that it will be tough to see the MDA Manifesto—and by implication the CMP Manifesto—through to adoption. This does not mean, however, that we should not hold CMP as a goal and work toward its realization and general acceptance. The benefits appear to be worth the costs.

3. Extreme non-programming:

We sometimes refer to CMP as XNP (eXtreme Non-Programming). Like XP (eXtreme Programming), programmers begin to code early in the development process, and the code is the model. Unlike XP, CM programmers do no programming at all—at least, they do not program in the traditional sense. Instead, the model is the code. XNP retains the advantages of XP and overcomes its disadvantages. A primary advantage of XP is that it allows clients, analysts, and programmers begin to see the application run immediately. XNP has this same advantage. XNP also retains other advantages typically attributed to XP, including responsiveness to changing client requirements, short development cycles resulting in improved productivity, and frequent client checkpoints and continuous client involvement. Primary disadvantages of XP are that it lacks overall analysis and has no overall design specification. XNP overcomes these disadvantages because the process focuses directly on analysis and specification, and the result of XNP is a design specification.

4. CMP in current practice:

Model-Driven Engineering (MDE), which is also referred to as Model-Driven Development (MDD) or Model-Driven Architecture (MDA), advocates the creation of software systems by model specification. As is the case for CMP, the models are abstract conceptualizations of particular domain concepts, rather than algorithmic specifications written in a high-level language, and conceptual modeling is the primary means of software production. In MDE, CASE tools generate code skeletons or, when enough detail is provided, they generate complete, deployable systems. Usually, however, only parts of the deployed system are fully generated. CMP requires full automation, including the full automation of enhancements and system evolution. Full automation avoids the prevalent pitfall of having conceptual diagrams

that are not synchronized with deployed systems. To the extent that MDE supports full automation, MDE and CMP are the same.

5. CMP status and outlook:

CMP is not just an academic dream. There are numerous commercially available model compilers, such as IBM Rational Rhapsody, the Olivanova tool suite from CARE Technologies, Netfective Technology Group's Blu Age, Obeo's Acceleo, the UWE UML Web Engineering platform, and WebRatio from Web Models to name just a few. As a specific example, consider the Olivanova technology, developed by *CARE Technologies* [CAR], S.A. Olivanova implements the OASIS approach to CMP [PHB92, PM07]. OASIS has a conceptual model with a precisely defined semantics that allows for a formal specification of all functionality needed for a final application. The conceptual model has four views that together completely specify an application for a management information systems: a static view, a dynamic view, a functional view, and a presentation view. A conceptual-model compiler translates modeling primitives into their corresponding software representations. The Olivanova technology automatically generates the final application from the specification of an OASIS model. The technology has two main components: a modeling tool called *Olivanova Modeler* and a model compiler called *Olivanova Transformation Engine*. The *Modeler* is a support tool that allows its users to specify an OASIS conceptual model and to verify that the conceptual model functions as expected. Then, when ready, the developer sends an XML representation generated by the *Modeler* to the *Olivanova Transformation Engine*, indicating the target implementation platform and some configuration parameters according to the selected platform. The *Transformation Engine*'s compiler automatically generates the source code of final applications, which is implemented for the selected platforms in a three-tier software architecture.

6. Additional readings:

Books by Dori [Dor09], by Embley [Emb98], by Embley and Thalheim [ET11], by Mellor and Balcer [MB02], by Morgan [Mor02], by Pastor and Molina [PM07], by Raistrick et al. [RFW+04], and by Rossi, Pastor, Schwabe, and Olsina [RPSO08] directly advocate CMP and explain how it works. Articles by Liddle, Embley, and Woodfield [LEW95, LEW00] describe model-equivalent languages and their role in CMP.

Books by Olivè [Oli07], by Papazoglou, Spaccapietra, and Tari [PST00], and by Thalheim [Tha00] focus more on conceptual modeling itself, but have a strong component that leads to CMP. A book by Harel and Politi [HP98] describes a CMP-styled approach to creating executable systems via statecharts. Another book by Ceri et al. [CFB+03] describes WebML, a CMP-styled approach to creating data-intensive web applications.

Many published articles discuss, argue for, and explain various aspects of CMP: formal specification of active objects along with rapid prototyping and object reification [PHB92], tunable formalism [CEW92], seamlessly combining multiple kinds of conceptual models [EJLW94], prototyping with conceptual models [JEW95], user interface modeling patterns [MMP02, PVE+07], and statecharts, both early work [HG97] and from a historical perspective [Har09].

Conceptual modeling itself has a long history. The book edited by Brodie, Mylopoulos, and Schmidt [BMS84] contains several articles that together provide an early look at the overall process leading to CMP. Proceedings of the International Conference on Conceptual Modeling [ERw] contain many articles that describe the

research and development of the field of conceptual modeling. An article by Thalheim [Tha09] summarizes and explains the conceptualization process in terms of an overall theory of conceptual modeling.

References

[BBI+04] G. Booch, A. Brown, S. Iyengar, J. Rumbaugh, and B. Selic. An MDA manifesto. *The MDA Journal: Model Driven Architecture Straight from the Masters*, pages 133–143, 2004.

[BMS84] M.L. Brodie, J. Mylopoulos, and J.W. Schmidt, editors. *On Conceptual Modelling: Perspectives from Artificial Intelligence, Databases, and Programming Languages*. Springer, New York, 1984.

[CAR] CARE-technologies web site. http://www.care-t.com/.

[CEW92] S.W. Clyde, D.W. Embley, and S.N. Woodfield. Tunable formalism in object-oriented systems analysis: Meeting the needs of both theoreticians and practitioners. In *Proceedings of the 1992 Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'92)*, pages 452–465, Vancouver, Canada, October 1992.

[CFB+03] S. Ceri, P. Fraternelli, A. Bongio, M. Brambilla, S. Comai, and M. Matera. *Designing Data-Intensive Web Applications*. Morgan Kaufmann Publishers, San Francisco, California, 2003.

[Dor09] D. Dori. *Object-Process Methodology: A Holistic Systems Paradigm*. Springer, Berlin, Germany, 2009.

[EJLW94] D.W. Embley, R.B. Jackson, S.W. Liddle, and S.N. Woodfield. A formal modeling approach to seamless object-oriented systems development. In *Proceedings of the Workshop on Formal Methods for Information System Dynamics at CAiSE'94*, pages 83–94, The Netherlands, June 1994.

[Emb98] D.W. Embley. *Object Database Development: Concepts and Principles*. Addison-Wesley, Reading, Massachusetts, 1998.

[ERw] ER web site. http://conceptualmodeling.org/.

[ET11] D.W. Embley and B. Thalheim, editors. *The Handbook of Conceptual Modeling: Its Usage and Its Challenges*. Springer, Heidelberg, Germany, 2011.

[Har09] D. Harel. Statecharts in the making: A personal account. *Communications of the ACM*, 52(3):67–75, March 2009.

[HG97] D. Harel and E. Gery. Executable object modeling with statecharts. *IEEE Computer*, 30(7):31–42, July 1997.

[HP98] D. Harel and M. Politi. *Modeling Reactive Systems with Statecharts: The Statemate Approach*. McGraw-Hill, Inc., New York, New York, 1998.

[JEW95] R.B. Jackson, D.W. Embley, and S.N. Woodfield. Developing formal object-oriented requirements specifications: A model, tool and technique. *Information Systems*, 20(4):273–289, 1995.

[LEW95] S.W. Liddle, D.W. Embley, and S.N. Woodfield. Unifying modeling and programming through an active, object-oriented, model-equivalent programming language. In *Proceedings of the Fourteenth International Conference on Object-Oriented and Entity-Relationship Modeling (OOER'95)*, pages 55–64, Gold Coast, Queensland, Australia, December 1995.

[LEW00] S.W. Liddle, D.W. Embley, and S.N. Woodfield. An active, object-oriented, model-equivalent programming language. In M.P. Papazoglou, S. Spaccapietra, and Z. Tari, editors, *Advances in Object-Oriented Data Modeling*, pages 333–361. MIT Press, Cambridge, Massachusetts, 2000.

[MB02] S.J. Mellor and M. Balcer. *Executable UML: A Foundation for Model-Driven Architectures*. Addison-Wesley-Longman Inc., Boston, Massachussets, 2002.

[MMP02] P.J. Molina, S. Meliá, and O. Pastor. User interface conceptual patterns. In *Proceedings of the 9th International Workshop on Interactive Systems. Design, Specification, and Verification (DSV-IS'02)*, Rostock, Germany, June 2002.

[Mor02] T. Morgan. *Business Rules and Information Systems: Aligning IT with Business Goals*. Addison-Wesley, Reading, Massachusetts, 2002.

[Oli07] A. Olivè. *Conceptual Modeling of Information Systems*. Springer, Berlin, Germany, 2007.

[PHB92] O. Pastor, F. Hayes, and S. Bear. OASIS: An object-oriented specification language. In *Proceedings of the 4th International Conference on Advanced Information Systems Engineering (CAiSE'92)*, pages 348–363, Manchester, United Kingdom, 1992.

[PM07] O. Pastor and J.C. Molina. *Model-Driven Architecture in Practice: A Software Production Environment Based on Conceptual Modeling*. Springer, New York, New York, 2007.

[PST00] M.P. Papazoglou, S. Spaccapietra, and Z. Tari, editors. *Advances in Object-Oriented Data Modeling*. The MIT Press, Cambridge, Massachusetts, 2000.

[PVE+07] I. Pederiva, J. Vanderdonckt, S. España, J.I. Panach, and O. Pastor. The beautification process in model-driven engineering of user interfaces. In *Proceedings of the 11th IFIP TC 13 International Conference on Human-Computer Interaction (INTERACT 2007)*, Rio de Janeiro, Brazil, September 2007.

[RFW+04] C. Raistrick, P. Francis, J. Wright, C. Carter, and I. Wilkie. *Model Driven Architecture with Exectuable UML*. Cambridge University Press, Cambridge, United Kingdom, 2004.

[RPSO08] G. Rossi, O. Pastor, D. Schwabe, and L. Olsina, editors. *Web Engineering: Modelling and Implementing Web Applications*. Springer, London, United Kingdom, 2008.

[Sel08] B. Selic. MDA manifestations. *The European Journal for the Informatics Professional*, IX(2):12–16, April 2008. http://www.upgrade-cepis.org.

[Tha00] B. Thalheim. *Entity-Relationship Modeling: Foundations of Database Technology*. Springer, Berlin, 2000.

[Tha09] B. Thalheim. Towards a theory of conceptual modeling. In *Advances in Conceptual Modelling - Challenging Perspectives (ETheCoM 2009 - First International Workshop on Evolving Theories of Conceptual Modeling)*, pages 45–54, Gramado, Brazil, November 2009.