Chapter 3.2.2
The STYLE Workbench: Systematics of Typed Language Environments

Ingrid Wetzel[1], Florian Matthes[2], and Joachim W. Schmidt[2]

[1] Universität Hamburg, Vogt-Kölln Straße 30, D-22527 Hamburg, Germany
[2] Technical University Hamburg-Harburg, Harburger Schloßstraße 20, D-21071 Hamburg, Germany

Summary STYLE is a method to systematically construct a customized development environment (CDE) for a given high-level data model by exploiting and extending an existing persistent programming environment along three dimensions.

On the functional dimension, polymorphic types and higher-order functions are used to extend the system functionality with new generic services and through the integration of external tools. On the operational dimension, orthogonal persistence is provided for application and meta data related to design objects. Finally, on the modal dimension, reflective capabilities of persistent programming environments make it possible to view data as executable code and vice versa. This is used in a CDE to switch dynamically between different modes of a design object.

The STYLE approach is illustrated using a concrete CDE implemented for an object-oriented data model exploiting and extending the Tycoon persistent programming environment. This STYLE workbench provides integrated graphical and textual modelling tools, generation support and data model animation.

1. Introduction and Overview

One of the remarkable features of persistent programming languages like PS-algol [3], Napier-88 (see Chapter 1.1.3) and Tycoon (see Chapter 1.1.1) is their rich set of data structuring facilities. Languages with orthogonal persistence and a sophisticated *type system* greatly simplify the task of implementing information systems that have to work with complex, long-lived bulk data structures (see Chapter 1.4.2).

In parallel to the development of persistent programming languages, research in the area of database and conceptual models led to the development of advanced *data models* (see, e.g. [1, 8, 11]) some of which have been integrated into database programming languages, such as Pascal/R [20], DBPL (see Chapter 2.1.2), Galileo [2] and Fibonacci (see Chapter 1.1.2).

The STYLE[1] approach [22] described in this paper clarifies the relationship between data models and type systems. STYLE is based on the assumption that high-level data models capture important application semantics that are beyond the scope of type systems (see section 2 for a more detailed discussion). Therefore, STYLE proposes a phase-oriented design process for information systems which involves transformation tools that bridge the gap between a given semantic (conceptual) model and the logical (computational) model underlying a typed programming language [7, 23]. The requirements on these tools are described in section 3.1. Due to the continuous evolution of data models, the set of transformation tools has to be tailored to the data model at hand.

[1] Style: Systematics of TYped Language Environments.

STYLE's main research contribution is a *systematic* method to construct a customized development environment (CDE) for a given high-level data model. A CDE exploits and extends an existing persistent programming environment (PPE) along three dimensions as discussed in section 3.2. On the functional dimension, polymorphic types and higher-order functions are used to extend the system functionality through the development of generic services and through the integration of external tools. On the operational dimension, orthogonal persistence is provided for application and meta data related to design objects. Finally, on the modal dimension, reflective capabilities of persistent programming environments make it possible to view data as executable code and vice versa. This is used in a CDE to switch dynamically between different modes of a design object.

In each of these three dimensions, the CDE tools are organized into four levels (kernel services, database functionality, data model support and data model usage) that exhibit different degrees of reusability. The resulting matrix architecture of a CDE is depicted in figure 3.1 and explained further in section 3.3.

In section 4 the STYLE approach is illustrated through a case study, the development of an integrated workbench for the high-level object-oriented data model OM1 [16] using the Tycoon system (see Chapter 2.1.4) as the underlying persistent programming environment. This CDE integrates multiple graphical and textual modelling tools, application generators and animation support which includes transactional programming and access to external servers.

2. Added Value through High-Level Data Models

This section discusses the main advantages of enriching a persistent programming environment by a high-level data model, namely:

- provision of domain-specific conceptual abstractions,
- system-enforced integrity beyond types, and
- enriched generic system functionality.

2.1 Domain-Specific Abstractions

Data models provide domain-specific abstractions that capture the essential modelling concepts required for the description of information systems. Data models emphasize the conceptual modelling of an application in terms of user concepts (e.g., classification, generalization, aggregation) instead of implementation abstractions (e.g., parameterization, modularization). These concepts are made available in a user-oriented language with a mnemonic syntax (with keywords chosen are close to natural language) that is easy to write and to understand. Many data models also specify an alternative or additional graphical representation.

Data models are a medium for the communication between software engineers and users during the modelling process and at the same time they should provide a precise specification of the system to be realized. Therefore, data models tend to blur the borderline between conceptual and computational perceptions of an information system.

2.2 System-Enforced Integrity beyond Types

Some data model concepts can be captured directly by type systems, others are beyond the scope of even the most sophisticated type systems. Nevertheless, it is

possible to provide language and system support for these concepts (like exception handling, run-time integrity checking or active rules) that leads to a system-enforced integrity beyond types. The following discussion of object-oriented data modelling concepts may explain this idea which is central to the STYLE approach.

An object has structural properties described in terms of attributes. An attribute can have a complex structure and it can reference other objects. A specialized object may inherit attributes of a more generic object, possibly with restrictions on their attribute domains. Classical type systems suffice to describe and enforce such structural object properties (perhaps lacking a user-oriented presentation). The fact that an object has an immutable identity during its full lifetime is also captured easily in a typed persistent language.

A class describes objects with the same set of properties. In the context of data models, a class serves several purposes which are partially incompatible with the notion of a typed class in object-oriented programming [22]. First, a class gives a structural description of objects (*intensional aspect*). In data models a class also denotes a set of objects with the same structure (*extensional aspect*). In some data models, the identification of an object is also based on its class (*identification aspect*). Finally, a class provides generic standard methods for creating, deleting, observing and modifying objects (*dynamic aspect*).

The interaction between the intensional and the dynamic aspects of a class leads to the difficult problem of how to handle the restriction of an attribute domain when an object is specialized. A domain restriction makes perfect sense from a modelling perspective but implies a covariant method specialization which leads to well-known difficulties in static type systems for which – up to now – no satisfactory solution has been found [9].

The extensional aspect of a class also goes beyond pure static type constraints. At first glance, it appears sufficient to declare a persistent bulk type variable to store the set of complex objects that make up the extent of a class. However, plain insert and delete operations on a class extent do not enforce an extensional aspect of classes, namely that the extent of a subclass is a subset of the extent of all its superclasses. This subset constraint can be enforced easily by a programming methodology where, for example, an object inserted in a class is automatically inserted into its (transitive) superclasses. Similarly, referential integrity constraints on class extents are introduced if objects in one class reference objects in other classes. Again, the verification of such inter-class constraints is beyond the scope of type systems.

Finally, some data models permit an object to migrate between classes that are not in a subclass relationship at all and to dynamically lose attributes during its lifetime without changing its object identity. Such operations can change dynamically the type of an object which has been bound to an identifier in a scope different from the scope of the update operation (for example, in another transaction). Therefore, such operations cannot be type-checked statically, even in languages with flexible subtyping or type matching rules. However, by enforcing a programming methodology that correctly propagates a type change of an object o to all scopes that possibly reference o, global application integrity can be ensured at run-time.

2.3 Enriched Generic System Functionality

Data models traditionally provide standard generic methods for objects of any class (create, delete, update, acquire attribute, lose attribute, ...). These methods have to satisfy (at least) the model-inherent integrity constraints. As discussed in the previous subsection, the semantics of these generic methods may depend on global

schema information and can therefore not be captured fully by type systems that rely on local type information only.

As described in [18], the code of such generic methods can be generated automatically based on the information provided by a data model. In a persistent programming environment, type-directed reflective programming techniques can be utilized to simplify this code generation process [22].

3. Data Modelling with STYLE

As explained so far, the STYLE approach supports a high-level data model by a customized development environment (CDE) that extends the functionality of a typed persistent programming environment.

Based on a classification of the tasks and tools in such a CDE given in section 3.1, the advantages of the STYLE approach which utilizes the PPE *itself* for the tool implementation are explained in section 3.2. Section 3.3 presents the generic matrix architecture underlying all CDEs that are built following the STYLE approach.

3.1 A Classification of Data Modelling Tools and Tasks

A CDE consists of a highly connected and smoothly integrated set of tools that automate work patterns which are typical for the design and implementation of an information system using a high-level data model. One can classify these tools based on the work patterns they support.

Modelling tools: On the data model level a CDE provides schema browsers and editors for graphical and textual modelling. Additionally, it assists in analyzing the status and the consistency of a schema through navigation tools (like class hierarchy, class association and integrity rule browsers) as well as local and global static schema checkers which are very similar to type checkers. Advanced CDEs automatically propagate local renaming and restructuring operations between different components of a large data model to ensure global schema consistency.

Generator tools: A CDE offers generators that map between different views (or modes) of a design object. On the data-model level, generators can ensure the consistency between textual and graphical schema representations. Code generators bridge the gap between data model and type system by generating type specifications enriched with procedural or declarative code to ensure the maintenance of high-level model-inherent and user-defined integrity constraints by the generic methods that insert, delete and update objects. Since the preservation of integrity constraints is central to the STYLE approach [15], the methods generated have transaction semantics with recovery functionality. Finally, generators can also be used to provide graphical or textual user interfaces that visualize typed persistent objects in a way that is consistent with the high-level data model.

Integration facilities: A CDE should integrate the modelling and generator tools described above and should provide analysis and visualization facilities for the whole design process. In particular, dependency information between design objects at different levels of abstraction should be recorded and, if possible, be exploited for automatic view and change propagation.

3.2 Specifics of the STYLE Approach

The systematic and flexible customization of development environments in STYLE exploits and expands the underlying PPE along three dimensions that should be orthogonal to each other to achieve maximum modelling and system flexibility [22].

Functional dimension: A rich language semantics including subtyping, polymorphism and higher-order functions enables the construction of libraries with generic services including type-safe wrappers for external design tool integration.

Operational dimension: Orthogonal persistence (object lifetimes independent of object types), distribution abstraction, multi-threading and orthogonal mobility of data, code and threads, greatly simplify the construction of CDEs.

Modal dimension: Modern persistent programming environments provide multiple, typed views onto code and data objects (concrete syntax, abstract syntax, intermediate code, executable code) and flexible bidirectional mappings between these modes as provided, for example, by extensible grammars (see Chapter 3.2.3) and reflective programming techniques [21]. This can be exploited in a design environment to realize flexible mode switches on design objects.

Each of these dimensions contributes to specific tasks in the construction of a CDE as follows.

Functional dimension for tool building: The positive consequences of using a strongly and polymorphically-typed PPE for CDE construction can be summarized as follows.

- Information system construction and design tool construction take place in the same language and system framework which facilitates code sharing and code re-use.
- The components of a CDE can be provided as an open set of typed libraries working on shared persistent data structures and not as a loosely coupled set of applications that exchange design information in an ad-hoc manner via files, command-line arguments or message queues. As sketched in figure 3.1, and discussed further in the next section, these libraries can be organized systematically into a matrix architecture.

Operational dimension for meta data management and analysis: Information system development is itself a data-intensive application [12] since it is necessary to store and retrieve bulk (meta) information about design objects (like graphical schemata, textual class definitions, or module interfaces) and design decisions. A design decision is an annotated relationship between design objects. For example, a design decision may relate a specification and several alternative implementations to an annotation that explains why a certain alternative was chosen.

The consequences of utilizing a PPE for CDE construction are obvious.

- Meta data of complex structure can be stored persistently without the need for complicated transformations into a canonical repository data format.
- Schema analysis and browsing tools can utilize tailored bulk data structures and high-level iteration abstractions to query meta data.
- It is possible to define declarative views on a design object that correspond to different levels of abstraction in a graphical schema or to multiple refinements of a design object.
- Change propagation can be realized by meta data analysis and updates exploiting integrity constraints defined for the meta data model.

Modal dimension for generation support: In STYLE, the graphical representation of a class, its textual description, its corresponding module and its GUI to visualize class instances are regarded as different modes of a single design object. Each mode has its own operations and tool support. Switches between different modes are implemented by parsers and generators that typically make use of the modal base services of a PPE.

A key characteristic of STYLE is the *integration* of the CDE tools as typed modules in an open persistent programming environment. This makes it possible to implement flexible cross-calls between tools for navigation purposes and to factor-out shared functionality like queries on meta data, generators or dependency tracking. A shared, strongly-typed and persistent meta data repository also contributes to a tight tool integration. Furthermore, external services can be integrated seamlessly by means of wrapper modules with strongly-typed interfaces.

3.3 The STYLE Matrix Architecture

On each of the three dimensions identified in the previous section, the CDE tools are organized into four layers (kernel services, database functionality, data model support and data model usage) that exhibit different degrees of reusability. The resulting matrix architecture of a CDE is depicted in figure 3.1.

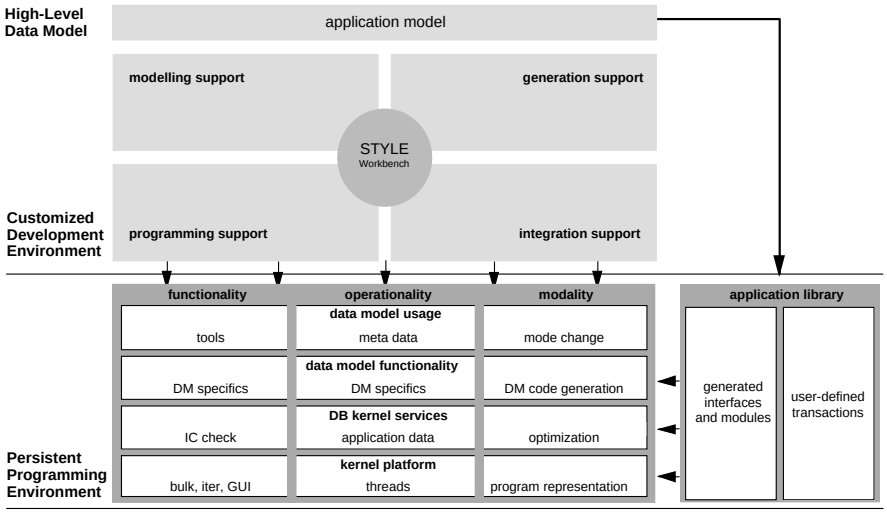

Fig. 3.1. The architecture of a STYLE customized development environment

The CDE is realized as a layered library of typed modules of the target persistent programming environment. The services of higher library layers are implemented based on services of lower library layers.

For example, the kernel services *bulk* and *iter* provide multiple bulk data structures (lists, sets, bags) and iteration abstractions (selection, projection, join) that

are utilized by a generic integrity checking layer *IC check* that implements predicative integrity checking on multiple bulk types. These services are specialized further in the next layer to support the inherent integrity constraints of a specific data model. The topmost layer is responsible for the presentation of these services to the user.

A similar layering applies to the operational and modal dimension. Some of the modules are simply wrappers for external tools (like text editors or graph editors) and external services (like SQL databases or GUI libraries).

A typical user of a CDE environment is not aware of the fine-structure of these tools but interacts directly with an integrated STYLE workbench that provides modelling, generation, programming and integration support.

An application described in a given high-level data model is transformed with the help of the CDE tools into a self-contained but open application library in the target language TL. This library consists of generated TL interfaces and modules which exploit the type system of the target language and utilize at run-time modules of the CDE itself to provide system integrity beyond typing. This dependency between applications and the CDE is indicated by horizontal arrows from the application library to the three lower levels of the CDE library.

4. A CDE for an Object-Oriented Data Model

In this section, the STYLE approach is illustrated by the description of a specific CDE, an integrated workbench for the high-level object-oriented data model OM1 [16] using the Tycoon system (see Chapter 2.1.4) as the underlying persistent programming environment.

4.1 Overview of the OM1 Data Model

The data model OM1 is based on concepts of semantic data models [11, 6] which emphasize the structural definition of objects. In addition, it provides additional constructs for formulating inter-object integrity constraints. Furthermore, OM1 is influenced by a formalization of object-oriented data models that supports both, structured values and objects [4, 5, 16, 19]. An application schema consists of a set of type and class definitions. Values are grouped into types describing immutable sets of values with a uniform structure together with operations defined on such values. Objects are abstractions of real-world entities and have an identity. This identity is independent of the values which are used to describe objects. Object identifiers are immutable during an object's lifetime. Objects with the same structure and behaviour are grouped into classes providing mutable collections as well as abstractions of individual objects. Relationships between objects are expressed by reference attributes. The description of multiple aspects or roles of the same object is modelled by simultaneous membership of an object in more than one class and by the migration of objects between classes.

OM1 has the usual model-inherent subclass and referential integrity constraints and supports the formulation of user-defined integrity constraints.

Application-specific object behaviour is expressed in the OM1 data model using a calculus derived from formal specification languages [10, 13] [17]. It is not necessary to specify standard query and update methods that satisfy the integrity constraints. Instead of this, integrity-preserving implementations for these methods are provided by the data model.

Class PackageTour

Relationships

Specialization
`isA` Tour `fix`
Dependencies
`deleteDependentOn` Hotel `with` removePackageTourWithHotel,
`deleteDependentOn` Flight ...

Structure

Attributes
`key` region : ▷ Region,
`key` town : ▷ Town,
hotel : ▷ Hotel,
flights : [forth : ▷ Flight, back : ▷ Flight]

Constraints

Static
HotelFlight:
(`this`.hotel.region = `this`.flights.forth.destAirport.region) ∧
(`this`.hotel.region = `this`.flights.back.depAirport.region),
(* the forth flight's destination airport and the return flight's
departure airport must belong to the same region as the hotel *),
...

Methods

Transactions
removePackageTourWithHotel
(packageTour : ▷ PackageTour date : Date) =
`begin`
ActualizedTour.modifyWithPackageTour(packageTour date);
removePackageTour(packageTour)
`end`
...

End

Fig. 4.1. A class definition in the object-oriented data model OM1

Figure 4.1 gives an example of a textual OM1 class definition. The class *Package-Tour* is part of a larger information system for travel agencies. The main classes of this application are flights, hotels and package tours (see also the nodes and arcs in the schema graph depicted in figure 4.2). Each package tour consists of a pair of back and forth flights together with hotel information. A user-defined integrity constraint states that the area specified for the flights has to match the area specified for the hotel.

4.2 User-Oriented Modelling Tools

The actual implementation of the OM1 CDE integrates the following modelling tools, some of which are visible in figure 4.2.

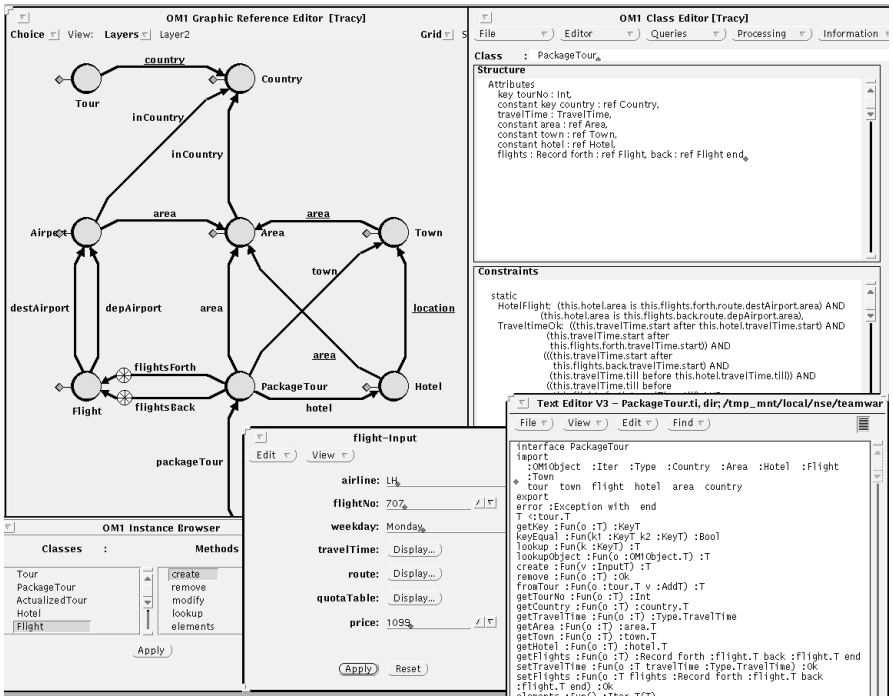

Fig. 4.2. Integration of components in a STYLE workbench

- A graphical schema editor is used to define class and type nodes with class reference arcs and inheritance links between classes. The editor supports views of different granularity.
- A class and type browser is used to select class and type names defined in a schema, for example, to invoke a text editor on a class.
- A textual class editor and a textual type editor are used to specify class components and type structures.

- Additional tools support the parsing, consistency checking and the generation of pretty-printed LaTeX documents for (parts of) an OM1 schema.
- A generator maps graphical OM1 class diagrams into corresponding textual OM1 class definitions.
- A toolbox provides additional local and global consistency checks and mechanisms to propagate changes in an OM1 schema (for example, class or attribute renaming). These algorithms can be invoked interactively from virtually all CDE tools.

4.3 Application Generators

```
interface PackageTour
import
  :OM1Object  :Iter  :Type  :Country  :Area  :Hotel  :Flight  :Town
  tour  town  flight  hotel  area  country
export
  error :Exception with  end
  T <:tour.T
  ...
  getKey :Fun(o :T) :KeyT
  keyEqual :Fun(k1 :KeyT k2 :KeyT) :Bool
  lookup :Fun(k :KeyT) :T
  lookupObject :Fun(o :OM1Object.T) :T
  create :Fun(v :InputT) :T
  remove :Fun(o :T) :Ok
  fromTour :Fun(o :tour.T v :AddT) :T
  getTourNo :Fun(o :T) :Int
  getCountry :Fun(o :T) :country.T
  getTravelTime :Fun(o :T) :Type.TravelTime
  getArea :Fun(o :T) :area.T
  getTown :Fun(o :T) :town.T
  getHotel :Fun(o :T) :hotel.T
  getFlights :Fun(o :T) :Record forth :flight.T  back :flight.T end
  setTravelTime :Fun(o :T  travelTime :Type.TravelTime) :Ok
  setFlights :Fun(o :T  flights :Record forth :flight.T  back :flight.T end) :Ok
  elements :Fun() :Iter.T(T)
  classInfo :Tuple name :String  end
end;
```

Fig. 4.3. A typed Tycoon implementation of the OM1 class *packageTour*

An OM1 class definition consists of structure, constraint and method specifications and is implemented by typed interfaces and modules in the Tycoon language TL (see figure 4.3). The TL implementation defines encapsulated objects with standard methods that satisfy the model-inherent and user-defined constraints. Access to the TL objects is only possible through the generated standard methods. User-defined class methods correspond to transactions of the information system and are built on top of the standard methods. As indicated in figure 4.3, the standard methods raise exceptions to signal integrity violations which can then be handled by the application code without breaking the encapsulation of the object.

The interfaces and modules created for class definitions extend the Tycoon environment with an application-specific library. The generated modules can be kept concise by utilizing pre-existing data-model-specific libraries of the CDE to factor out repeating tasks from the generated code. The provision of data-model-specific libraries is termed *situative lifting of the target language* in the STYLE approach.

The cooperation between TL code generators and the generic CDE services can be exemplified by the integrity checking task of a CDE. For a specific OM1 integrity constraint (e.g., referential integrity between two classes), preconditions and delayed conditions have to be generated for the insert and delete methods of the classes involved. Instead of hard-wiring these conditions into the method's implementation code, a generic TL library for constraint handling is utilized. It provides dynamic collections of constraints (implemented as bulk collections of function values in TL) that can be attached dynamically to methods. As a consequence, the constraint handling is fully separated from the OM1 interface and module generation process. Global OM1 constraints are transformed into boolean TL predicates which are then attached dynamically to the relevant methods.

The main benefit of this approach is *locality*. A local update of a single class definition leads to a local update of the corresponding TL class implementation and a reinitialization of the constraint database. No other method code needs to be recompiled.

4.4 Animation Support

An animation component provides a generic database browser to create, delete and update individual objects (instances of classes) and to navigate through the persistent object base following the (set-valued) associations between objects. The animation tool uses the generated application modules to guarantee the enforcement of integrity constraints specified in the data model. The type-specific visualization code is also generated automatically.

4.5 Integration with Tycoon Transaction Code

Arbitrary user transactions can be implemented on top of the generated class implementations using the algorithmically-complete persistent programming language TL. For example, a user-defined method *removeHotel* can extend the functionality of the *hotel.remove* operation by rebooking all package tours prior to the removal of a hotel. See [14] for a discussion on how generic methods and user-defined transaction code have to interact to maintain the consistency in an object base.

The language TL also provides mechanisms to enable interoperability of OM1 applictation code with existing (legacy) database systems like Ingres, Oracle and ObjectStore.

4.6 Application Development Using the Integrated Prototype

Figure 4.4 sketches the main modelling steps supported by the OM1 customized development environment. First, the structural schema information is modelled using the graphical editor.

Based on this information the structural part of the textual class definitions is generated. Interactive queries at the schema level can be utilized to detect incomplete schema information, e.g. references to undefined classes. Integrity constraints

and user-defined class methods are also added at this stage using a structured-text editor. Navigation facilities to referenced and referencing classes simplify the formulation of inter-class integrity constraints.

The programming layer is automatically populated with the generated class interfaces and modules implementing the standard methods. The application programmer implements the user-defined methods on top of these interfaces. If necessary, user-defined methods can also override standard methods.

The animation component supports the validation of an OM1 model by populating a persistent object base with sample objects and by invoking standard and user-defined methods on these objects. The visualization code generated for the animation can also be reused and adapted for use after the prototype phase.

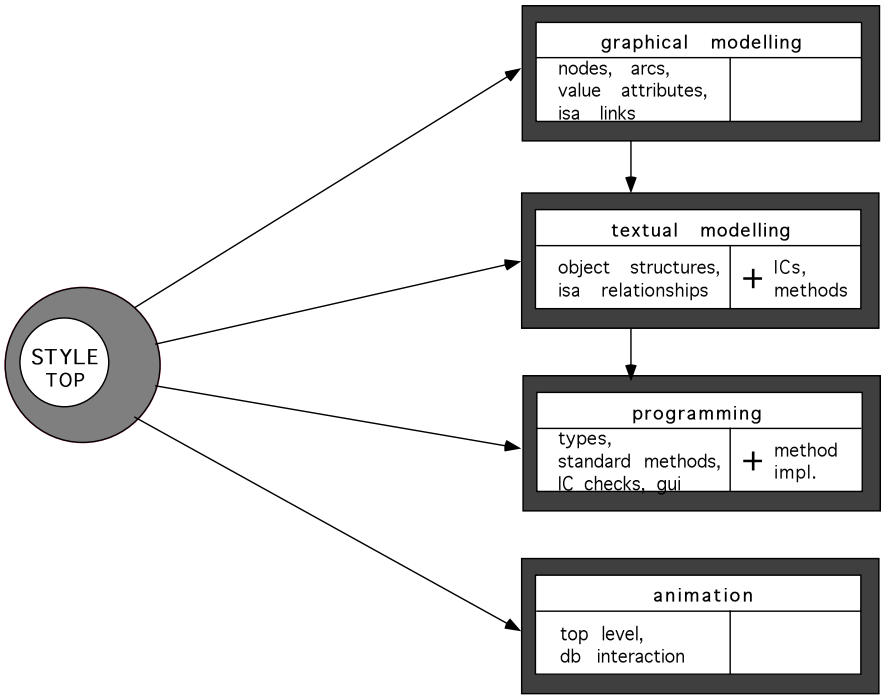

Fig. 4.4. Tasks and tools involved in the OM1 modelling process

5. Concluding Remarks

The customized development environment for the OM1 data model clearly demonstrates that persistent programming environments are a suitable platform for both the construction of data-intensive applications utilizing a high-level data model as well as for the construction of tightly integrated design and modelling tools.

The distinction between functional, operational and modal requirements was of great use in the organization of the STYLE tool infrastructure. It remains to be verified how much of this infrastructure can be re-used for the construction of other CDEs.

Acknowledgement This research is supported by ESPRIT Basic Research, Project FIDE, #6309.

References

1. J.-R. Abrial. Data semantics. In J.W. Klimbie and K.L. Koffeman, editors, *Data-Base Management*. North-Holland, Amsterdam, 1974.
2. A. Albano, L. Cardelli, and Orsini R. Galileo: A strongly-typed, interactive conceptual language. *ACM Transactions on Database Systems*, 10(2):230–260, 1985.
3. M.P. Atkinson, K.J. Chisholm, and W.P. Cockshott. PS-algol: An algol with a persistent heap. *ACM SIGPLAN Notices*, 17(7), July 1981.
4. C. Beeri. A formal approach to object-oriented databases. *Data and Knowledge Engineering*, 5:352–382, 1990.
5. C. Beeri. New data models and languages – the challenge. In *Proceedings of the Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 1992.
6. A. Borgida, J. Mylopoulos, and J. Schmidt. The TaxisDL software description language. In M. Jarke, editor, *Database Application Engineering with DAIDA*, pages 65–84. Springer-Verlag, 1993.
7. R.P. Brägger, A. Dudler, J. Rebsamen, and C.A. Zehnder. Gambit: An interactive database design tool for data structures, integrity constraints and transactions. In C.A. Zehnder, editor, *Database Techniques for Professional Worskstations*, pages 65–96. Department Informatik, ETH Zürich, Switzerland, September 1983.
8. M.L. Brodie, J. Mylopoulos, and J.W. Schmidt, editors. *On Conceptual Modelling, Perspectives from Artificial Intelligence, Databases, and Programming Languages*. Springer-Verlag, 1984.
9. W. Cook. A proposal for making eiffel type-safe. In *ECOOP 90 Proc. European Conference on Object Oriented Programming*, 1989.
10. E.W. Dijkstra and C. S. Scholten. *Predicate Calculus and Program Semantics*. Springer-Verlag, 1989.
11. R. Hull and R. King. Semantic database modeling: Survey, applications, and research issues. *ACM Computing Surveys*, 19(3), 1987.
12. M. Jeusfeld, M. Mertikas, I. Wetzel, Jarke. M., and J.W. Schmidt. Database application development as an object modelling activity. In *Proceedings of the Sixteenth International Conference on Very Large Databases, Brisbane, Australia*, August 1990.
13. G. Nelson. A generalization of Dijkstra's calculus. *ACM Transactions on Programming Languages and Systems*, 11(4):517–561, October 1989.
14. K.-D. Schewe. *Specification of Data-Intensive Application Systems*. PhD thesis, Technische Universität Cottbus, 1994.

15. K.-D. Schewe, J.W. Schmidt, D. Stemple, B. Thalheim, and I. Wetzel. A reflective approach to method generation in object oriented databases. Rostocker Informatik Berichte Nr. 13, Fachbereich Informatik, Universität Rostock, Germany, 1992.
16. K.-D. Schewe, J.W. Schmidt, and I. Wetzel. Identification, genericity and consistency in object-oriented databases. In J. Biskup and R. Hull, editors, *Proceedings of the International Conference on Database Theory*, volume 646 of *Lecture Notes in Computer Science*, pages 341–356. Springer-Verlag, October 1992.
17. K.-D. Schewe, J.W. Schmidt, I. Wetzel, N. Bidoit, D. Castelli, and C. Meghini. Abstract machines revisited. FIDE Technical Report Series FIDE/91/11, FIDE Project Coordinator, Department of Computing Sciences, University of Glasgow, Glasgow G128QQ, March 1991.
18. K.-D. Schewe, B. Thalheim, J.W. Schmidt, and I. Wetzel. Integrity enforcement in object-oriented databases. In U. Lipeck, editor, *Proc. 4th Int. Workshop on Foundations of Models and Languages for Data and Objects, Volkse, Germany, October 19–22*, 1992.
19. K.-D. Schewe, B. Thalheim, and I. Wetzel. Foundations of object-oriented database concepts. Informatik Fachbericht FBI-HH-B-157/92, Fachbereich Informatik, Universität Hamburg, Germany, November 1992.
20. J.W. Schmidt. Some high level language constructs for data of type relation. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data, Toronto, Canada*, August 1977.
21. D. Stemple, T. Sheard, and L. Fegaras. Linguistic reflection: A bridge from programming to database languages. In *Proceedings 25th Annual Hawaii International Conference on System Sciences*, pages 46–55, 1992.
22. I. Wetzel. *Programmieren mit STYLE: Über die systematische Entwicklung von Programmierumgebungen*. PhD thesis, Fachbereich Informatik, Universität Hamburg, Germany, July 1994.
23. I. Wetzel, K.-D. Schewe, J.W. Schmidt, and A. Borgida. Specification and refinement of databases. In M. Jarke, editor, *Database Application Engineering with DAIDA*, pages 283–318. Springer-Verlag, 1993.