

Developing Interoperable Software For Differing XML Schemata

Thuy-Linh Nguyen, School of Business and Informatics, Australian Catholic University,
Locked Bag 4115 Fitzroy MDC VIC 3065 Australia.
Email: linh.nguyen@acu.edu.au. Phone: +61-3-9953-3168. Fax: +61-3-9953-3775.

Abstract

The advent of XML [XML04] has made it possible for various XML schemata, or document types, to co-exist. Questions then arise about how developers can write, or what support they can have to write, interoperable XML schemata and interoperable software and tools associated with them. These XML schemata may originate from the same XML schema but have evolved and diversified over time. This paper describes Life Design and LifeWeb, work done in this area.

Keywords: interoperability, document type, XML schema, evolvability

1 Introduction

The advent of XML [XML04] has made it possible for various document types, or XML schemata, to co-exist. Questions then arise about how developers can write, or what support they can have to write, interoperable XML schemata and interoperable associated software and tools associated with them. These XML schemata may originate from the same XML schema but have evolved and diversified over time. This paper describes Life Design and LifeWeb [NGU01a&b], work done in this area.

In this work, an XML schema is implemented as a object-oriented type system where an element type (with its attributes) corresponds to a type (with its attributes and methods). (Thus an element type is augmented with behaviour defined by the methods of its corresponding type.) A document is equivalent to an instance of a type system (that corresponds to the document's schema). As XML schema essentially defines element types, its evolution can be considered in terms of the evolution of the element types, and interoperability between evolving XML schemata can also be considered in similar terms. Other features of XML schema such as namespace may be omitted for the purpose of this paper. More details about mapping an XML schema to an object-oriented type system may be found in [NGU01a].

Using such mapping, the following pairs of terms may be used interchangeably in this paper: *XML schema* and *type system*, *document* and *system instance*, *element type* and *type*, *element* and *object*. The term *system schema* may also be used interchangeably with the term *type system*.

2 Life Design

Life Design is a framework for an object-oriented and evolvable system and is gleaned from (what is believed to be) natural evolution. Evolvability necessarily implies interoperability. This section briefly explains the part of Life Design that has enabled interoperability between systems.

2.1 Interoperability

A key factor in obtaining interoperability in Life Design is by satisfying the conformity constraint from object-orientation [RUM91]. In this constraint, if type T_1 is derived from type T_0 (ie. T_1 is a subtype of T_0) then T_1 is conformed with T_0 . This derivation is carried out through the inheritance relationship [RUM91] in which T_1 necessarily includes all attributes, methods, and relationships (collectively referred to as *features*) defined in T_0 , and may have extra ones.

We extend the notion of type conformity to include also (object-oriented) system conformity. To this end, if type system S_1 is derived from type system S_0 then S_1 is conformed with S_0 . This derivation is also carried out through the inheritance relationship in which S_1 includes (inherits) all types (with their relationships) in S_0 , and may have extra ones that must be derived from some type(s) in S_0 . These extra types may have relationships with other types that may or may not be in S_0 such that the formation of

such relationships do not change the definition of any existing types (see also below).¹ In the examples in Figure 1, systems S_4 and S_5 are both derivatives of S_1 , while S_2 and S_3 are not. The latter two (S_2 and S_3) are directly derived from S_0 , the parent of S_1 . Clearly conformity implies interoperability. In Figure 1 for instance, S_0 , S_1 and S_4 are interoperable with one another, and so are S_0 , S_1 and S_5 .

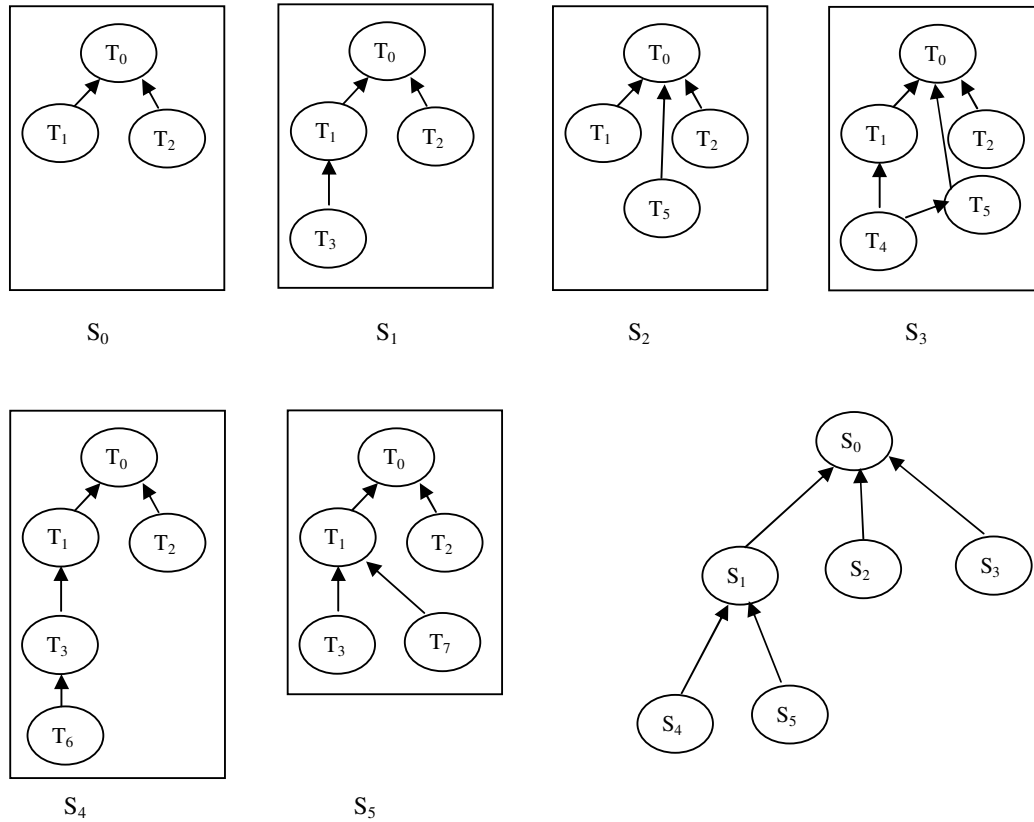


Figure 1. Type and system conformity

During the development of a type system S through various versions $\{S_0, S_1, S_2, \dots, S_n\}$, if we can ensure that any new version S_i of S is conformed to some older version S_{i-j} ($j < i$) of S , then S_i is interoperable with S_{i-j} and any parents of S_{i-j} . It is noted that S_i is not interoperable with the children of S_{i-j} which are not S_i , if any. They only share a common parent system, which is S_{i-j} . When two systems are not conformed (thus not interoperable) with each other, we say that they belong to two distinct *species*.² It is not possible to obtain full interoperability between systems that branch out to different species. But as long as conformity is maintained, design and code from the parent can be reused. In the system inheritance tree in Figure 1, all nodes (representing systems) that can be traced in one single direction along the inheritance path are interoperable to one another, and belong to the same species.

In Life Design, during the evolution process, conformity among members in one species can be satisfied by observing the following rules:

¹ There are also other constraints for system evolvability in Life Design. This paper only discusses constraints relating to interoperability. Interested readers are referred to [NGU01a].

² As the evolution process in Life Design resembles the natural one described by Darwin [DAV98, DAR59], the term *species* has been used to denote branches of systems that share a common root but are no longer interoperable with each other. This structure, which is depicted in the last diagram in Figure 1, is similar to the tree of life described in [DAR59].

- (i) The definition of an existing type (its features) must not be changed during its whole lifetime
- (ii) If it is necessary to change the definition of an existing type, such as adding, removing or modifying its features, a new type (with the desired features) must be developed and introduced into the system through an appropriate inheritance relationship.

These rules prohibit the adding and removing of attributes and methods from an existing type. A type is also not allowed to change its supertype(s), or relationships with other types that are owned, used, or contained, by it. (That is, the other types are part of its definition.) These rules ensure that conformity of the existing system is preserved, and any new types introduced are conformed to some existing type in the system, thus conformity of the whole evolving system is still preserved. For instance, in system S_5 in Figure 1, in order to remove a feature in type T_3 , a new type T_7 must be derived from T_1 , the parent of T_3 , and added to the system through that inheritance relationship.³

2.2 Scalability

Observation of the above rules does raise a question. Since any change to a type involves adding a new type, will the system grow indefinitely? What if a new type is meant to replace an existing type?

The short answer to the above questions is that, it is possible to remove a type provided that (i) it does not have any corresponding objects, and (ii) it is not used in the definition of any type that must remain in the system. (That is, it is not a supertype of any other type, and is not associated with any type such that it is part of the definition of the other type, where the other type must stay in the system after its removal.) In other words, the type to be removed must not have anything depending on it, both at the object (instance) and type (schema) levels. These two conditions form what we call *dependency clearance*.

It is possible to satisfy dependency clearance, although this usually requires careful system design through all stages of system evolution. We will now look at how dependency clearance has been achieved in other literature, and how it can be achieved in Life Design.

2.2.1 Dependency clearance at the object level

To free a type of any corresponding objects we need to migrate all of its objects to the new type. In database schema evolution (DBSE), this is done by techniques such as *screening*, *conversion* and *filtering* [PET97]. All of these techniques explicitly *coerce* instances to reflect changes of the schema. Coercion takes place at different times and in different ways in different techniques. For instance, in screening objects are coerced when they are accessed by applying (possibly multiple) conversion program(s) on them. Screening suffers performance penalty during object access and can increase system overhead to keep track about which objects still need to be updated. The conversion technique converts affected objects immediately on each schema change. This causes processing delay during the schema modification phase, but not during object access. The filtering approach does not propagate changes to objects, but creates a different version of a schema for each schema change, and the old objects remain with the old version while new objects are created for the new version. Filtering introduces overhead for maintaining and coordinating between different versions.

Life Design uses the concepts of *birth* and *death* (in natural evolution) to control the *multiplication* and *extinction* of a type (migrating objects to a new type and removing an existing type). Briefly, in Life Design, a system instance (such as a document) has a *life expectancy*, captured in two attributes:

- *relativeExpiration*, which specifies the amount of time that the system instance can live from the time it is last used (accessed)
- *absoluteExpiration*, which specifies the maximum amount of time that the system instance can live.

³ In some case these rules may be relaxed when the change involves only adding (and not removing) some features of a type. However, when automatic evolution is required, they should be observed, as can be seen in the following sections.

If a system instance has not been accessed (used) for a *relativeExpiration* period of time, or if it is still used but has reached the *absoluteExpiration* time, it will be destroyed. In the first case, it will be destroyed forever. In the second case, it will be immediately recreated (*reincarnated*) with any necessary updates to reflect the most recent evolutionary change in the system schema. This way, the system instance as well as the objects contained in it can be cast into (migrated to) the most recent versions of their types, and their original versions released (to be removed later).

Casting a system instance into a new schema version is always possible since either the new version is conformed to the old one, or it does not matter if it is not. This is because by dependency clearance constraint at the type (schema) level (see below), any missing types in the new schema version must not have any associated objects. Thus the lack of those types do not have any effect when the system instance is cast from the old into the new schema version (see Figure 2).

Casting an object into a new type is always possible if the new type is a derivative of the old type since the former conforms to the latter. When the new type is not derived from the old type, the object of the old type need *recapturing*. An object of the new type should be created to meaningfully replace the object of the old type. In bottom-up evolution where changes start at the object level first, such an object may be created by a human user (with semantic constraints defined by a human developer when applicable) and inserted into the system (see [NGU01a&b]). In top-down evolution where changes are imposed by developers at the schema level, instructions about the formation of such an object may be given in program code. For instance, in Figure 2 suppose T_4 is meant to substitute T_3 , and T_4 does not have all the attributes defined in T_3 but have some of its own. The instructions may specify how to convert the extra T_3 attributes to T_4 attributes, or simply discard them. The dependency clearance constraint at the schema level (see below) ensures that the type to be removed (the old type) does not have any types dependent on it. The replacement of objects of type T_3 by objects of type T_4 therefore does not affect the functioning of any other types or the whole system. Discussions about bottom-up and top-down evolutions can be found in [NGU01a&b].

This recreating and casting (*reincarnation*) process, which is repeated many times over the entire population of system instances of a given schema (type system), eventually replaces all occurrences of the old version of the schema with the new one. At this point of time, all the old types (in the old version of the schema) become free of dependency at the object level, and the old schema can be removed (but not the old types yet – see details below).

This approach controls the multiplication and extinction of a (version of a) system schema (type system) in a decentralised and asynchronous manner. It can be implemented relatively simply, for example, by running a low priority thread that quietly checks and updates system instances, and removes old versions of system schemas. The major advantage of this approach over comparable techniques in DBSE is that it does not suffer performance penalty as screening or conversion, and has less overhead than filtering.

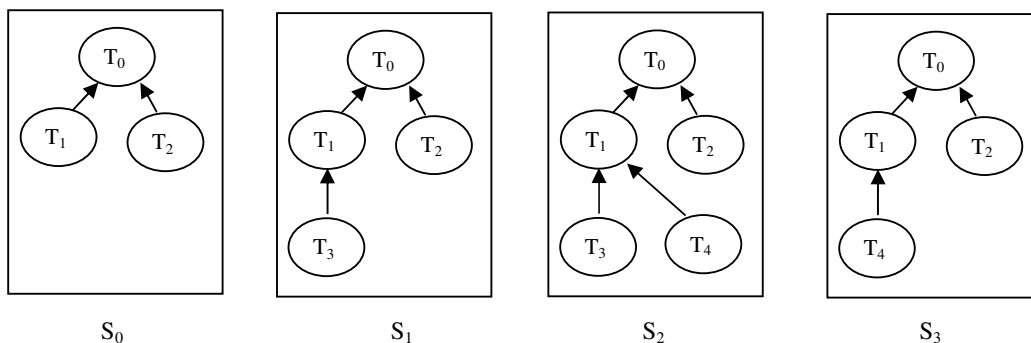


Figure 2. Migration of objects and system instances. T_4 is replacing T_3 .

- **Migration from T_3 to T_4 need recapturing.**
- **Migration of system schemas is always possible: Migration from S_1 to S_2 is possible since S_2 conforms to S_1 . Migration from S_2 to S_3 is also possible since T_3 now has no corresponding objects.**

2.2.2 Dependency clearance at the type (schema) level

Database schema evolution generally does not place any constraint on a type before it can be removed [BAN87]. (Even the object migration described above in DBSE is only *re-active*, that is, it happens *after* schematic changes, not before.) Thus the notion of dependency clearance at the schema level does not exist in DBSE. For instance, a type can change its supertype or a relationship with a type that forms part of its definition. This gives the freedom to remove or add any type through any relationship, but can generate a ripple effect to all affected types and objects, and lead to expensive overhead due to the need to maintain schema and data consistency throughout the system.

The constraint of dependency clearance at the schema level can help avoid this cost. Satisfying this constraint may be simple or complicated depending on where the type is in the inheritance tree. In some cases it may not be necessary to remove a type.

- (i) If the type is a leaf in the inheritance tree and it is not owned, used or contained in any other type, then it already satisfies the constraint
- (ii) If the type is a leaf in the inheritance tree, and it is owned, used or contained in some other type(s), then we either cannot (should not) remove it, or have to remove it together with the other type(s). The first option underlines the importance of early design choices – some may not be practically possible to undo. The second option requires careful design and involves a recursive operation. It can be as expensive if recursion loops are deep
- (iii) If the type is not a leaf in the inheritance tree (ie. it has one or more subtypes) and it is not owned, used or contained in any other type, then it is usually not necessary to remove it. Its presence in the system schema but not in any system instances is simply like that of an abstract type. Interestingly this phenomenon is often encountered in nature where traces of ancestors' design can be found in descendants' genes but are not realised [NGU01a]
- (iv) If the type is not a leaf in the inheritance tree and it is owned, used or contained in some other type(s), then the situation is similar to (ii) or (iii).

Note that when a type is removed from a system schema S_n , a new schema, say S_{n+1} , is formed. S_{n+1} is no longer interoperable with S_n – that is, S_{n+1} has started a new species. S_{n+1} however is still interoperable with the parent of S_n . If S_{n+1} is meant to replace S_n , all instances of S_n must have been migrated to S_{n+1} and S_n will become extinct. This case is often applicable in the evolution of one single system (such as one XML schema). If S_{n+1} is not meant to replace S_n , both of them still have their own instances and the two species simply co-exist, having a common root. This case is often found in the development of differing systems deriving from a common programming interface (such as XML schemata for different domains but sharing the same DOM (Document Object Model) [W3C] interface).

In short, Life Design controls the evolution process in such a way that a system can only be changed on a component (type) basis around the inheritance relationship. This ensures interoperability between versions of system schemata in the same species. Types can be added and also removed so that the system is scalable during evolution, and new species can be formed, branching out from some common root.

2.3 Automatic evolution

The constraints described above make it possible to automate the evolution process to some extent using relatively simple techniques and with fairly low overhead. When evolution happens on one single machine or set of machines for which we have read and write permissions directly or indirectly, the following tasks can be automated:

- (i) Carrying out evolution tasks such as migrating objects to new types, migrating system instances to new system schemata, adding new types/schemata to or removing obsolete types/schemata from, the runtime system, and possibly also from the file system if required
- (ii) Keeping track of changes in system instances and schemata, and ensuring consistency of the evolving system when evolution tasks are carried out

With some careful design, the first task above can be done declaratively. An implementation of such design will be described briefly in the next section.

We note that the following tasks cannot be automated, at least not with the currently available technology:

- (i) Initial as well as evolving system design
- (ii) Implementation of the initial system
- (iii) Implementation of any new type
- (iv) Giving evolution instructions, such as what type should be added, removed, or replaced by what. (These instructions can often be given as attribute values stored together with the evolving type.)⁴

3 LifeWeb

This section describes some relevant implementation experiences of LifeWeb, a system that implements Life Design. LifeWeb is an object-oriented data model for the Web document system. It captures a document in its three intrinsic aspects: structure, presentation and content. Thus in its core LifeWeb has four types: Document, StructuralComponent, Presentation and Material, as shown in Figure 3.

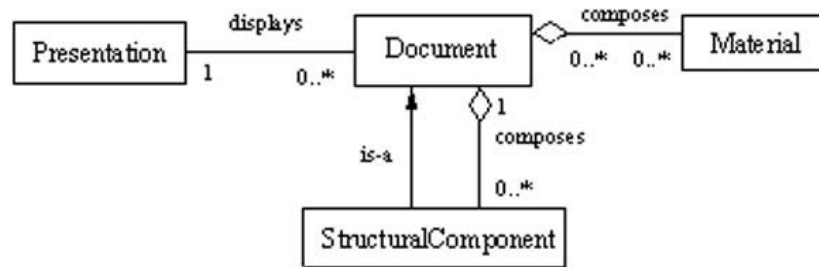


Figure 3. Core LifeWeb data model

The LifeWeb data model is represented in XML, that is, its types are XML element types, and LifeWeb itself is defined as an XML schema. Aggregation is represented by nesting elements in one another, association by referencing between elements, and inheritance by a special attribute *superclass*, known to the system. All LifeWeb documents must be valid XML documents, ie. they must declare the LifeWeb Schema in them, so that the schema can be retrieved and edited if necessary (see [NGU01a] for more details).

In LifeWeb Schema, every element type has two attributes:

- *behaviour*, which captures the full class (type) name of the implementation of the element type. For instance, the string *behaviour="au.edu.monash.sd.lifeweb.LWText"* stored in element type `<text>` means that the element type `<text>` is implemented as the (Java) class *LWText* in the package *au.edu.monash.sd.lifeweb*
- *replacedBy*, which specifies the element type name that is to replace the current element type, if any. (The value of this attribute may be an empty string, which means the current element type is not being replaced. It may also be the string "null", which means the current element type is to be removed without replacement.)

The core implementation of LifeWeb, which is written in JavaTM, consists of the LifeWeb engine, which implements the LifeWeb data model, and a LifeWeb document parser (class name LWParser),

⁴ Actually this task has already been automated in LifeWeb, where evolution is also driven by user input at the instance level (bottom-up), not just by developers at the schema level (top-down) [NGU01a&b]. This paper however, discusses mainly the top-down approach.

which extends an XML parser. This core system can be intergrated with different front-ends to build applications specific to the LifeWeb Schema. For instance, when intergrated with a Web server, it can be used to deliver LifeWeb documents to the client. When intergrated with a GUI (Graphical User Interface), it can serve at the back-end to build an authoring tool for creating and managing LifeWeb documents. Figure 4 shows the basic architecture of LifeWeb, intergrated with a Web server.

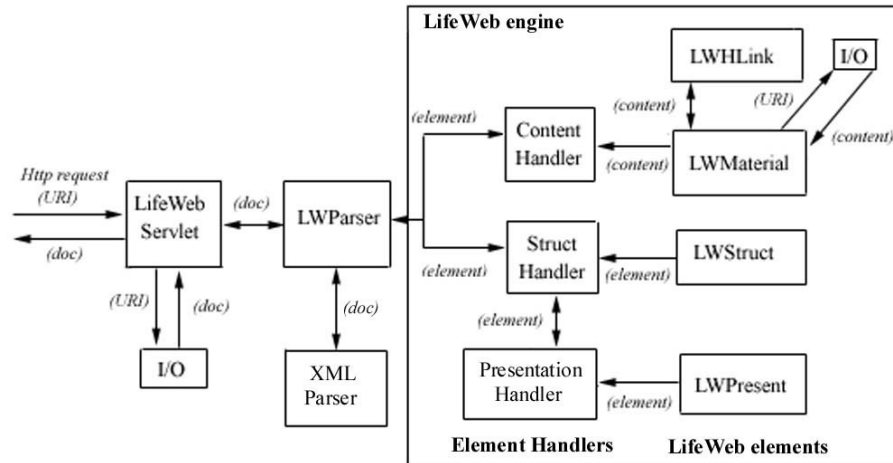


Figure 4. Basic LifeWeb system architecture (intergrated with a Web server)

New element types may be declaratively added to the system at any time by editing the LifeWeb Schema (to form a new version of the schema). In top-down evolution, this is simply done manually by a human developer. In bottom-up evolution this is done automatically by the system when some evolutionary threshold is satisfied (see [NGU01a&b] for more details). When the system is deployed (for instance, a LifeWeb document is requested), the information stored in the *behaviour* attribute for each element type in the LifeWeb Schema is retrieved to dynamically instantiate the class that implements that element type, and register it to the LWPParser. (This can be done using the Java dynamic class instantiation feature [DEI98].) Thus, any declarative changes made to the LifeWeb DTD will be functionally and dynamically incorporated into the runtime system.

In top-down evolution, existing element types may be declaratively marked to be removed or replaced at any time by manually editing the corresponding attribute *replacedBy* in the LifeWeb Schema (to form a new version of the schema). In bottom-up evolution, element types are removed or replaced automatically by the system when some evolutionary threshold is satisfied (thus the attribute *replacedBy* is not necessary). (See [NGU01a&b] for more details.) A low priority thread periodically checks LifeWeb documents for their *life expectancy* and performs any necessary and permissible updates. These include such operations as replacing an old type/schema by a new type/schema (in a LifeWeb document), removing an old schema (from the file system), or removing an old type (from the schema and also from the file system if required). (The last operation effectively creates a new branch of LifeWeb schemata.) Similar to the addition of a new element type, removing an element type from LifeWeb schema (declaratively) also removes its corresponding class implementation functionally and dynamically from the runtime system.

4 The implications

Clearly the approach described in this paper promotes evolvability which implies a number of important software qualities such as reusability, manageability, maintainability, extensibility and interoperability. Where a generic, core model can be identified, a variety of systems can be derived from it, all sharing, ie. reusing, the same core model. If such a core model can be standardised, or even better, a specific programming interface can be standardised, then its implementation can be widely shared and reused. Components of future systems derived from it may also be exchanged or shared among these systems, since they conform to the same core model or programming interface. The Document Object Model (DOM) [W3C] is a clear example of the benefits of a standard generic core model for all document types. In light of this standardisation effort, it is arguable that further generic core model for document types may still be derived from DOM, for instance, to capture the “document

characteristics” common to all document types. It may also be arguably possible to have a generic core model for each specific domain upon which document types can be used.

This approach places some restrictions on the way a system or an XML schema can be evolved. These restrictions however, make it possible to establish a well-defined evolutionary path. Systems evolve along this path in a cohesive manner, and remain manageable, maintainable and interoperable. This allows for the evolutionary tasks that should be automated to be automated relatively simply and efficiently.

5 Conclusion

Schema evolution and system interoperability are well-established areas in Computer Science [BAN87, PET97, JAR98]. Applications of these areas in systems represented in XML however, are not yet known to us. By using the conformity constraint in object-orientation and adding the dependency constraint, Life Design is able to define a clear evolutionary path where systems evolve strictly on a component (type) basis around the inheritance relationship. This and the technique to migrate objects and system instances to new types permits some evolutionary tasks to be automated more simply and efficiently than can be done in DBSE. System consistency can be maintained automatically at both levels instance and schema. Besides, evolution in Life Design can happen in either direction, bottom-up or top-down; in DBSE it can only be top-down.⁵ Finally the capturing of the behaviour of an element type allows for some evolutionary tasks to happen simply declaratively.

References

- [BAN87] Banerjee J., et al, 1987. Semantics and implementation of schema evolution in object-oriented databases. *In: Proceedings of the ACM SIGMOD Annual Conference on Management of data*, San Francisco, CA USA, 27-29 May 1987. ACM Press, 311 – 322
- [DAR59] Darwin, C., Burrow, J. W., ed., 1859. *The Origin Of Species By Means Of Natural Selection*, Penguin Books
- [DAV98] Davis, P., 1998. *The Fifth Miracle: The Search for the Origin of Life*, Allen Lane, The Penguin Press
- [DEI98] Deitel & Deitel, 1998. *Java^m: How to program*. 2nd edition. Prentice-Hall Incorp
- [JAR98] Jarke, M. et al, 1998. Meta Modeling: A Formal Basis for Interoperability and Adaptability. *In: B Kramer, M Papazoglou, H-W Schmidt, eds., Information System Interoperability*. Research Studies Press Ltd, John Wiley & Sons Co, 229-263
- [NGU01a] Nguyen, T-L, 2001. An Object-Oriented Data Model for Evolvable Web Systems. PhD Thesis. Monash University
- [NGU01b] Nguyen, T-L, 2001. LifeWeb And Web Evolvability. *In: Proceedings of International Conference, Internet Computing 2001*, Las Vegas, USA, July 2001.
- [XML04] *Extensible Markup Language (XML) 1.0* (2004) [online]. World Wide Web Consortium. Available: <http://www.w3.org/TR/2004/REC-xml-20040204/> [Accessed 7 Mar 2006].
- [W3C] World Wide Web Consortium [online] Available: <http://w3.org> [Accessed 7 Mar 2006].

⁵ Bottom-up evolution is more similar to that in nature, in which more evolutionary tasks can be automated and the system can reshape itself when it evolves. Interested readers are referred to [NGU01a].