# Representing Architectural Evolution

Erdogmus, Hakan

# Representing Architectural Evolution

Hakan Erdogmus

Institute for Information Technology, National Research Council
Montreal Road, Building M-50, Ottawa, Ontario, Canada K1A 0R6
*Hakan.Erdogmus@nrc.ca*

## Abstract

Software engineers informally use block diagrams with boxes and lines to express system architectures. Diagrammatic representations of this type are also found in many specification techniques. However, rarely are architectural documents containing such representations systematically maintained; as a system evolves, architectural documents become obsolete, and the design history of the system is ultimately lost. Additionally, box-and-line representations used in these documents do not possess a precise semantics invariant across the different techniques that rely on them. This paper addresses expression of system evolution at the architectural level based on a formal model of box-and-line diagrams. The formal model (a) provides semantic uniformity and precision; and (b) allows evolutionary steps to be represented as structural transformations. Interesting classes of such transformations are characterized in terms of the underlying operators. With these tools, the architectural evolution of a system is captured as a directed acyclic graph of baselines, where each baseline consists of a system of box-and-line diagrams, and is mapped to a successor baseline by a set of structural transformations. It is also shown how familiar design concepts—such as extension, abstraction, and structural refinement—can be formalized in simple terms within the framework developed.

## 1 Introduction

Block diagrams with boxes and lines are commonly used in software engineering to informally express system architectures. Numerous specification and modeling techniques for designing distributed systems also rely on them; examples include ROOM [15], SDL [14], Modechart [7], DSL [19], and Lotos [6]. Moreover, this type of system structuring is implicit in software architecture description languages [17]—such as Wright [1], Aesop [3], Unicon [16], Rapide [8], and ACME [4]—as well as their predecessors, module description languages [12]. The common aspect of all of these techniques is their adoption of the familiar *component-connection-configuration* paradigm [9], whereby large systems are constructed by interconnecting smaller subsystems. This paradigm focuses on what may be referred to as runtime architectures, such as expressed by an object or module connection (interaction) diagram, rather than on source-code architectures, such as expressed by a UML[1] class diagram [13].

Although they are quite intuitive, box-and-line diagrams do not possess a precise semantics uniform across the specification techniques that employ them. The structural semantics is often intertwined with the behavioral semantics of the particular technique used. Consequently, architectural documents written in different specification techniques are difficult to share.

A common semantic model is a good start,

[1]UML is a trademark of Rational Software Corporation.

but ignores one important aspect: system evolution. Evolution is inevitable since both requirements and technology change rapidly. Unfortunately, architectural documents may, over time, become obsolete; often, they are produced once, or are not maintained systematically. Furthermore, these documents are rarely put under configuration management. Consequently, the design history of the system is eventually lost. However, this information can be valuable during maintenance, redesign, and future extensions for large and long-lived systems, especially when employee turnover is high. Since keeping track and recording architectural changes may be laborious, automation is desirable. A formal approach supports automation through explicitness, precision, and rigor.

To address system evolution formally at the architectural level, this paper defines a set-theoretic model based on box-and-line diagrams; introduces an abstract, flexible notion of transformation for these diagrams; and identifies several different kinds of such transformations to express common architectural changes. As such, a general framework emerges in which to represent the architectural evolution of a system at a high level through a sequence of purely structural transformations. The framework allows the formalization of familiar design concepts—such as extension, structural refinement, and abstraction—in set-theoretic terms.

As a motivation, consider the following example adapted from Selic et al. [15]. Figure 1 depicts the architectural evolution of a PBX system during its modeling stage. The initial model is very simple: it only considers two users (telephones) and disregards administration functions. The *TelHandler* components shields the rest of the system from future changes to the supported types of telephone equipment. The *Call* component encapsulates the capabilities related to calls. At the next step, the model is extended to include a component *Admin* for administration functions. *TelHandler* is replaced by an extended version *TelHandler′*, which can interface with the administration component. Then the two *TelHandler′* components are abstracted into a single component, *TelHandlers*, to be able to later generalize the system to accommodate $N$ users. Next, the *TelHandlers* component is replaced by a generic component *GenericTelHandler* that can can handle up to $N$ users through a single interface. Similarly, the *Call* component is replaced by a generic call handler (*GenericCall*) that can handle several calls at once. At the same time, a connection subsystem is added so that the call handler would no longer be concerned with the details of establishing, maintaining, and releasing connections. Finally, the connection subsystem is refined into a component that isolates switching capabilities and another that interfaces with the call handler. Thus the evolution of the system from the initial to the final model is recorded as a sequence of box-and-line diagrams related through conceptual design operations (namely, extension, abstraction, structural refinement, and component replacement). This representation contains valuable information that would otherwise be lost, and helps in the comprehension and communication of the design choices made during the evolution of the system. Apart from being used in model development to keep track of architectural changes as illustrated in this small example, such representations can also be used during re-design or re-engineering to communicate and control design decisions. Similarly, they can be used for configuration management of components at a high level in component-based system development, as supported by emerging component technologies such as JavaBeans [18][2].

The purely structural view of architecture has its limitations: it does not address all aspects of architectural abstraction. In particular, functional properties, performance, and reliability concerns are left out. Neither are the underlying design rationales made explicit by this view. Additionally, for some systems, the box-and-line architecture is trivial, or does not convey any useful information about the organization of the system. A purely structural model may be insufficient to relate the *intended* architecture to the *as-implemented* architecture and to prevent architectural drift. All of these issues are important from the point of view of system evolution, and may be treated in a larger framework. Our treatment is lim-

[2] JavaBeans is a trademark of Sun Microsystems, Inc.

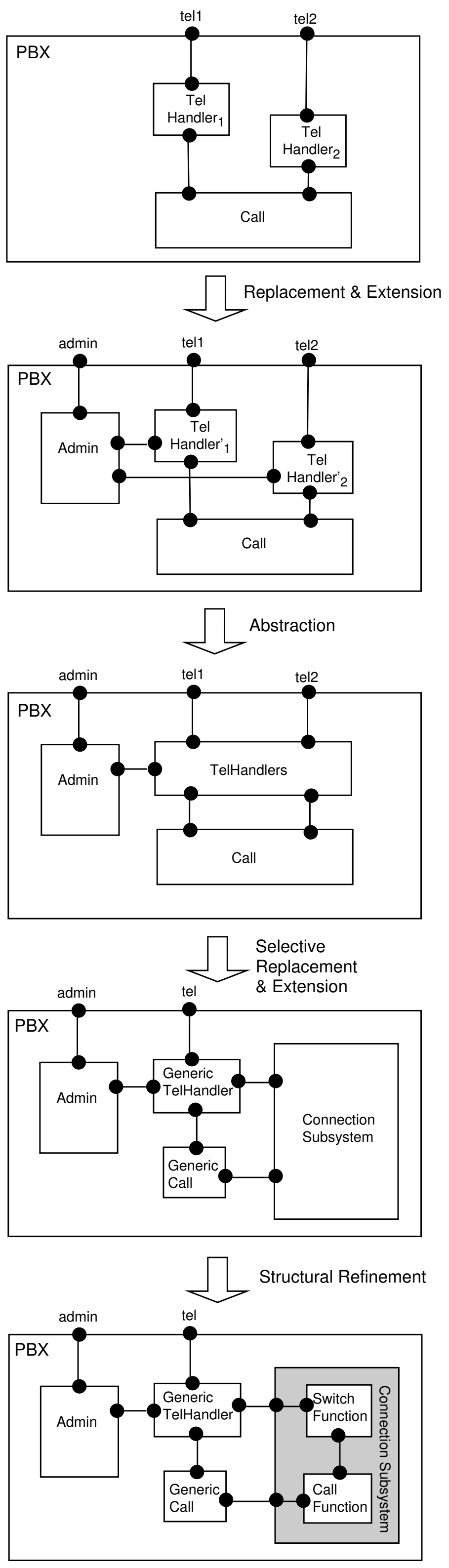

Figure 1: Model evolution example.

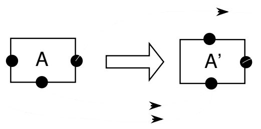

Figure 2: A structural transformation.

ited to those aspects that can be addressed by structure alone, and there is much insight to be gained by focusing on the structural aspect.

A central concept introduced in the paper is that of a structural transformation. A structural transformation defines a semantic correspondence between a source system and a target system through a mapping on their interface ports, as shown in Figure 2. The structural transformation does not imply that the two systems provide equivalent external functionalities relative to some behavioral model. It simply states that, under the specified mapping, the system on the right will effectively replace the system on the left, regardless of any variation in the actual overall external functionality. The mapping defines for each interface port of the system on the left, a corresponding interface port of the system on the right such that the semantic role played by the image port subsumes the semantic role played by the source port. The transformation does not imply that the external functionality of the system on the right subsumes that of the system on the left. This information often cannot be determined based on structure alone, and is meaningful only relative to a behavioral model. The overall external functionality of a system is not simply the sum of its interface ports and its constituent components. It is the behavioral model that determines the external, or black box, behavior by assigning functionality to the lowest level modules and defining a composition construct to infer the external functionality of a composite module from those of its constituents. This latter subject is addressed by numerous semantic theories of concurrency and programming languages, and is beyond the scope of this paper. The reader is referred to Milner's seminal work [11] for a classical example.

Given the notion of structural transformation, the architectural history of a system can be recorded as a directed acyclic graph (DAG),

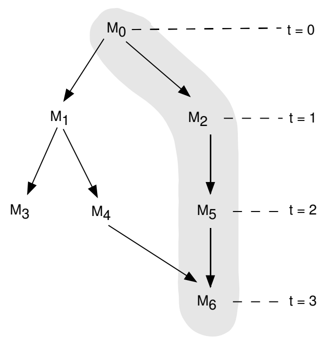

Figure 3: An evolution graph. The nodes represent different baselines and the arrows represent the *deltas* between subsequent baselines. The shaded area identifies a path in the (directed acyclic) graph with four evolution steps labeled from $t = 0$ to $t = 3$.

where each node represents a baseline that consists of a system of box-and-line diagrams. A baseline is mapped to its successor by a set of structural transformations. Thus the transformations represent *deltas* that establish semantic correspondences between neighboring pairs of baselines along a given path in the evolution graph (Figure 3). The evolution graph is usually a tree. It becomes a DAG if two baselines on different paths are merged into a common baseline in their subsequent versions.

The model has a deficiency. Although the structural transformations defined allow merging of interfaces in subsystems, they do not permit interface splitting or refinement. Interface refinement is common practice, but is difficult to formalize as a homomorphic transformation. Further structuring of interfaces is neither sufficient nor necessary to formalize the concept since interface refinement affects connections in a context-dependent way. The affected connections have to be specified in every context in which the interface-refined subsystem appears. This in turn makes interface refinement a problematic operation. The treatment of interface refinement is planned as future work, and as such, is not addressed in this paper. See Section 6 for further discussion.

The rest of the paper is organized as follows: In Section 2, the general concept of a module based on box-and-line structures and the associated formal model are introduced. Section 3 discusses structural relations on modules and states a fundamental closure property of these relations. The notion of structural transformation is introduced in this section. Section 4 defines a set of relevant operators on box-and-line structures. Each of these operators underlies a particular class of structural transformations. Representation of architectural histories is addressed in Section 5, where some of the operators introduced in Section 4 are related to familiar design concepts. A summary and discussion can be found in Section 6.

## 2 Module Systems

The term *module* refers to the basic unit of abstraction used to construct systems. We depict the structure of a module in terms of an enclosing box, inside of which may be a set of boxes interconnected by solid lines, as illustrated in Figure 4. Such diagrams are commonly referred to as box-and-line or block diagrams.

A module has two associated views:

1. The black box view specifies the *external structure* of the module, and involves a fixed interface. In a box-and-line diagram, the interface is indicated by the enclosing box.

2. The clear box view specifies the *internal structure* of the module in terms of a collection of *instances* of other modules, called *components*. The components are interconnected in a fixed configuration. In a box-and-line diagram, components are represented by boxes inside an enclosing box, and connections by solid lines between the inner boxes.

### 2.1 Attributes of a Module

The most fundamental attribute of a module is its *interface*. The interface consists of a set of *ports* which collectively specify the boundary through which an instance of that module can be interconnected with other modules to form a larger system. Ports are depicted by dark circles in box-and-line diagrams.

Each component of a module is specified as an *instance* $C_i$ of some other module $C$. Then

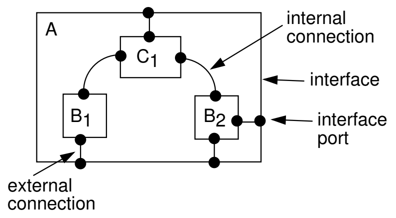

Figure 4: The box-and-line structure of a module $A$. Interface ports are distinguished from one another by their relative placement on module boundaries.

$C$ is said to be the *type* of the component $C_i$, written $Typ(C_i) = C$.

Some components are connected to each other through their interface ports, while others may be connected to the interface ports of the enclosing module. We write $p.C_i \smile_M q.D_j$ if in module $M$, the interface port $p$ of a component $C_i$ is connected to the interface port $q$ of another component $D_j$. Such a connection is called an *internal connection.* The internal connection relation $\smile_M$ is symmetric; hence $p.C_i \smile_M q.D_j$ implies $q.D_j \smile_M p.C_i$. We write $p \frown_M q.D_j$ if the interface port $q$ of a component $D_j$ is connected to the interface port $p$ of the enclosing module $M$. Such a connection is called an *external connection.*

Figure 4 illustrates these concepts.

## 2.2 Formal Model

A *module system* $\mathbf{M}$ is a triple $\langle Mods, Ports, Str \rangle$, where

- *Mods* is a set of *module names*;
- *Ports* is a set of *port names*; and
- for each module $M \in Mods$, $Str(M)$ specifies the architecture of $M$ in terms of a *box-and-line structure* over $\mathbf{M}$.

A *box-and-line structure* (or *bl-structure*) $S$ over a module system $\mathbf{M}$ is a quadruple $\langle Intf_S, Cmps_S, \smile_S, \frown_S \rangle$, where

- $Intf_S$ is the *interface* of $S$;
- $Cmps_S$ is the set of *components* of $S$;
- $\smile_S$ is the set of *internal connections* of $S$; and
- $\frown_S$ is the set of *external connections* of $S$.

We have

- $Intf_S \subseteq Ports$ is a finite set;
- $Cmps_S \subseteq Mods \times Nat$ is a finite set where each component $C_i$ ($i \in Nat$) is an *instance* of some module $C \in Mods$;
- $\smile_S \subseteq (Ports \times Cmps_S) \times (Ports \times Cmps_S)$ is a finite, symmetric, irreflexive relation such that $p.C_i \smile_S q.D_j$ implies $p \in Intf_C$ and $q \in Intf_D$.
- $\frown_S \subseteq Intf_S \times (Ports \times Cmps_S)$ is a relation such that $p \frown_S q.D_j$ implies $q \in Intf_D$.

For a module $M \in Mods$, we often say "the interface of $M$" to mean "the interface of $Str(M)$," and similarly for the components and the sets of connections of $M$. By abuse of notation, *Mods* and $\mathbf{M}$ will be used interchangeably: $M \in \mathbf{M}$ should be understood as $M \in Mods$. Accordingly, given $Str(M) = S$, we simply write $Intf_M$, $Cmps_M$, $\smile_M$, and $\frown_M$ to denote $Intf_{Str(M)}$, $Cmps_{Str(M)}$, $\smile_{Str(M)}$, and $\frown_{Str(M)}$, respectively.

A module $N$ is a *submodule* of another module $M$, written $N < M$, if $M$ has a component $N_i$ (of type $N$). The transitive closure of the relation $<$ is called the *dependence relation*, and is denoted by $<^*$. For a module system to be well defined, the graph of the dependence relation must be free of cycles. This property is assumed throughout the paper.

A module $M$ is called a *leaf module* if $Cmps_M = \emptyset$. Hence the internal structure of a leaf module in unspecified. A leaf module occupies a leaf node in the dependence graph of a module system.

# 3 Structural Relations

To support the formal model introduced in Section 2, a proper semantic equivalence relation must be defined on modules. It would not be very useful to semantically equate two modules if and only if they are identical. A weaker concept is needed. Then what kind of relation would be suitable to decide whether there is any practical reason to distinguish between two given modules in a module system based

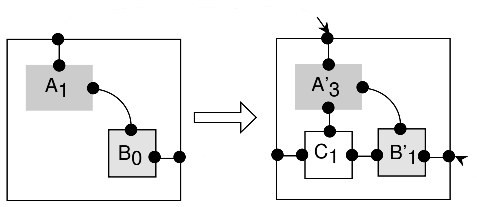

Figure 5: A bl-homomorphism.

on their structures alone? An isomorphism on bl-structures underlies such a relation. We will refer to this isomorphism as a *bl-isomorphism.* The notion gives rise to the sought semantic relation when an extra property is satisfied.

A weaker, but independently useful concept is that of a *bl-homomorphism.* A bl-homomorphism maps the bl-structure of a module to that of a second module, while preserving both the internal and the external configuration of the first, as illustrated in Figure 5. A bl-isomorphism is simply a bi-directional bl-homomorphism in which the underlying mappings are reversible.

The notion of a bl-homomorphism can be used to express the fact that a given module may structurally evolve into another module such that for each interface port, component, and connection of the first module, the second module has a corresponding interface port, component, and connection, respectively, with a similar (or larger) semantic role.

However, a mapping of internal structures (components, internal ports, and connections) is in general not necessary to represent the architectural evolution of a module. A mapping of its external structure (interface ports) is sufficient, since the internal structure can be subjected to arbitrary changes as the module evolves, whereas a certain degree of semantic consistency should be maintained in the external structure. The notion of *structural transformation* captures this less restricted form of evolution. By a set of such transformations, it is possible to represent a one-step evolution of a system of modules. We will refer to a set of structural transformations as a *transform relation.*

If a module evolves through a sequence of arbitrary changes to its internal structure, then all modules that depend on that module will also evolve simultaneously. This principle is referred to as *substitutivity.* Therefore, the implied architectural evolution of the dependent modules can also be captured by a transform relation. However rather than being arbitrary, the changes to the internal structure of these dependent modules are dictated by the changes to the internal structure of the modules on which they depend. Here the notion of bl-homomorphism comes into play: the structural transformation of one dependent module to another is guided by an underlying (surjective) bl-homomorphism.

## 3.1 Structural Homomorphisms

Given two bl-structures $S$ and $R$, a triple $\langle \sigma, \gamma, \Sigma \rangle$ is called a *bl-homomorphism* from $S$ to $R$ if

1. $\sigma\colon \mathit{Intf}_S \longrightarrow \mathit{Intf}_R$ is a function from the interface of $S$ to the interface of $R$;

2. $\gamma\colon \mathit{Cmps}_S \longrightarrow \mathit{Cmps}_R$ is a function from the components of $S$ to the components of $R$;

3. $\Sigma \overset{\text{def}}{=} \{\sigma_{C_i} \mid C_i \in \mathit{Cmps}_S\}$ is a set of functions such that for the component $C_i$ of $S$, the function $\sigma_{C_i}\colon \mathit{Intf}_{\mathit{Typ}(C_i)} \longrightarrow \mathit{Intf}_{\mathit{Typ}(\gamma(C_i))}$ maps an interface port of the type of $C_i$ to an interface port of the type of $\gamma(C_i)$ in $R$; and

4. the homomorphism preserves the connections:

   - $p.C_i \smile_S q.D_j$ implies $\sigma_{C_i}(p).\gamma(C_i) \smile_R \sigma_{D_j}(q).\gamma(D_j)$
   - $p \frown_S q.D_j$ implies $\sigma(p) \frown_R \sigma_{D_j}(q).\gamma(D_j)$.

A bl-homomorphism is illustrated in Figure 5. In the figure, the mapping $\gamma$ is color coded and the mapping $\sigma$ is indicated by dashed arrows. The mappings in $\Sigma$ are not shown, but can be inferred from the physical placement of the components' interface ports.

The mathematical concept of bl-homomorphism is fundamental. It underlies the design concept of structural inheritance (without component exclusion), such as implemented in the ROOM [15] technique, whereby a module is allowed to inherit its structure from another module and extend it if necessary.

## 3.2 A Structural Equivalence for Modules

The definition in the previous subsection gives rise to a *bl-isomorphism* if all the functions involved ($\sigma$, $\gamma$, and the $\sigma_{C_i}$ of $\Sigma$) are bijective (one-to-one and surjective). The isomorphism preserves component types only if the function $\gamma$ preserves component types—that is if $\mathit{Typ}(\gamma(C_i)) = C$, for every $C_i \in \mathit{Cmps}_S$.

Now we can define a semantic relation on bl-structures based on the notion of bl-isomorphism. Denote this relation by $\approx$. For two bl-structures $S$ and $R$ such that $\mathit{Cmps}_S \neq \emptyset \neq \mathit{Cmps}_R$, we write $S \approx R$ if $S$ and $R$ are related by a bl-isomorphism that preserves component types. Note that $\approx$ is an equivalence (reflexive, transitive, and symmetric), and as such, can suitably play the role of a semantic relation for bl-structures. It is illustrated in Figure 6. What $\approx$ essentially accomplishes is to abstract away from module and interface port names as well as component indices.

The relation $\approx$ is considered to be the strongest of all interesting semantic relations on bl-structures. Thus if two bl-structures are related by this relation, not only their structures are isomorphic, but also they should provide exactly the same external functionality. Therefore, if $\mathit{Str}(M) \approx \mathit{Str}(N)$, then $M$ and $N$ are considered to be practically indistinguishable, and the two are safely interchangeable in all contexts. Note, however, that $\mathit{Str}(M) \not\approx \mathit{Str}(N)$ does not necessarily imply that $M$ and $N$ have different external functionalities. So $\approx$-equivalence is a sufficient, but not a necessary, condition for behavioral equivalence.

By stipulating that $\mathit{Cmps}_S \neq \emptyset$ and $\mathit{Cmps}_R \neq \emptyset$, the possibility of $\approx$ equating two different leaf modules is avoided.

## 3.3 Transform Relations

The ability of a module to undergo an architectural change at a point during its lifetime is formalized by the notion of structural transformation. The architectural change may represent a specialization, a (structural) refinement, an extension, or some other conceptual design operation.

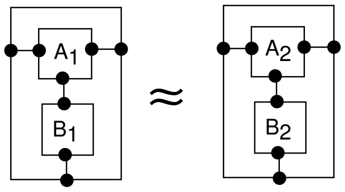

Figure 6: Two $\approx$-equivalent bl-structures.

Given two module system $\mathbf{M}$ and $\mathbf{N}$, a *(structural) transformation* over $\mathbf{M} \times \mathbf{N}$ is a triple $\langle M, \sigma, N\rangle$, where $M \in \mathbf{M}$, $N \in \mathbf{N}$ and $\sigma\colon \mathit{Intf}_M \longrightarrow \mathit{Intf}_N$ is a function which associates an interface port of $N$ with every interface port of $M$.

A set $\rightsquigarrow$ of transformations over $\mathbf{M} \times \mathbf{N}$ is called a *transform relation* over $\mathbf{M} \times \mathbf{N}$. The shorthand notation $M \overset{\sigma}{\rightsquigarrow} N$ is used to mean $\langle M, \sigma, N\rangle \in \rightsquigarrow$, and is read as "$\rightsquigarrow$ transforms $M$ to $N$ under $\sigma$." When $\rightsquigarrow$ is clear from the context, we simply say "$M$ transforms, or evolves, to $N$ under $\sigma$."

Transform relations can be used to represent the architectural evolution of an entire system in terms of structural transformations across sets of modules. This subject will be discussed in Section 5.

Consider a *base transform relation* $\rightsquigarrow$ which identifies the transformations that are by some external means known to be semantically sound. That is, if $M \overset{\sigma}{\rightsquigarrow} N$, it is given that for each interface port $p$ of $M$, the corresponding interface port $\sigma(p)$ of $N$ is locally capable of assuming a similar semantic role as $p$ in any context in which $N$ replaces $M$. Here we leave the exact meaning of the term "similar" uninterpreted since the concept varies for each different specification technique. In general, it refers to some kind of interface or port type compatibility. The key point is that the base relation captures some arbitrary changes to the structure of a system, and assumes that the semantic soundness of these changes is established outside the model.

However, a change to a part of a system

may affect other parts of the the system due to interdependencies between the parts. If a module evolves through a sequence of transformations, then all modules that depend on that module should also evolve simultaneously through a similar sequence of transformations. This principle is known as *substitutivity.* More precisely, substitutivity states that if a surjective bl-homomorphism[3] relates two modules such that each component in the source module transforms to the homomorphic image of that component in the target module, then the first module transforms to the second.

The substitutivity principle is illustrated in Figure 7. As an example, suppose module $M$ has a component of type $C$ (i.e., $M$ depends on $C$) and $C$ transforms to $D$. Replace that component by a component of type $D$. Then $M$ transforms to the module resulting from the substitution of $D$ for $C$ in $M$.

Given a base transform relation $\rightsquigarrow$, a minimal transform relation always exists that (1) contains $\rightsquigarrow$, and (2) is closed under substitutivity. Let $\rightsquigarrow_*$ denote this relation. Note that $\rightsquigarrow_*$ is unique if every module in $\mathbf{M}$ is unique up to the equivalence $\approx$ (for every $M, N \in \mathbf{M}$, $M \neq N$ implies $\mathit{Str}(M) \not\approx \mathit{Str}(N)$). Thus, $\rightsquigarrow_*$, or *the substitutive closure of* $\rightsquigarrow$ can be defined as the smallest relation that satisfies the following properties:

1. $M \overset{\sigma}{\rightsquigarrow} N$ implies $M \overset{\sigma}{\rightsquigarrow}_* N$ ($\rightsquigarrow_*$ contains $\rightsquigarrow$).

2. if

   2.1 $\mathit{Cmps}_M \neq \emptyset$ ($M$ is a non-leaf module), and

   2.2 there exists a surjective bl-homomorphism $\langle \sigma, \gamma, \Sigma \rangle$ from $M$ to $N$ such that $C \overset{\sigma_{C_i}}{\rightsquigarrow}_* \mathit{Typ}(\gamma(C_i))$ for every component $C_i \in \mathit{Cmps}_M$

   then $M \overset{\sigma}{\rightsquigarrow}_* N$.

Note that by property 2, $\mathit{Str}(M) \approx \mathit{Str}(N)$ implies $M \overset{\sigma}{\rightsquigarrow}_* N \overset{\sigma^{-1}}{\rightsquigarrow}_* M$, for some $\sigma : \mathit{Intf}_M \longrightarrow \mathit{Intf}_N$.

[3]A bl-homomorphism is surjective if all the mappings involved (the functions $\sigma$ and $\gamma$, and the functions in $\Sigma$) are surjective.

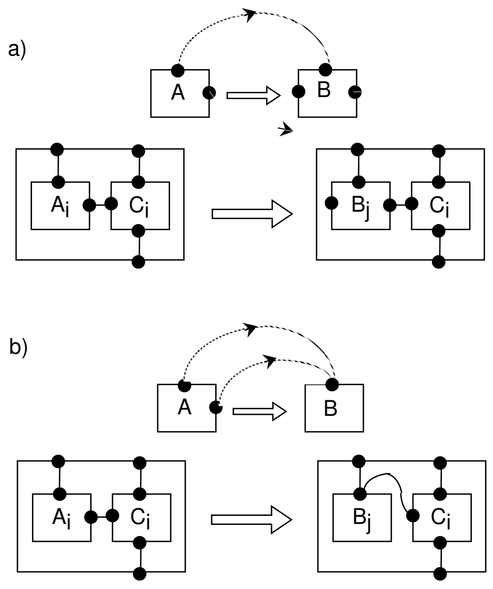

Figure 7: Substitution.

## 4 Structural Operations

We depart from the premise that the high-level architecture of a system often does not evolve in a completely arbitrary and chaotic way. Certain transformational patterns are applied over and over again. Although this premise is not validated by hard empirical evidence, we believe that most software designers would agree with it. In other words, the evolutionary changes are usually guided by well defined operators that transform the structure of the parts to which they are applied in particular ways. It is not sufficient for such operators to return a new structure. Their application must also return a mapping which specifies a semantic correspondence between the old and the new structures.

Formally, a *structural operator* is a partial function on bl-structures (it is more convenient to define the notion on bl-structures than on modules). When such an operator is applied to a bl-structure that satisfies the operator's preconditions, it returns a new bl-structure over the same module system, together with a semantic correspondence function from the interface ports of the operand to those of the resulting bl-structure.

Let *op* denote a structural operator, and $S$

a bl-structure over $\mathbf{M}$. Then $op(S)$ returns a pair $\langle R, \sigma\rangle$ where $R$ is a bl-structure over $\mathbf{M}$ and $\sigma\colon \mathit{Intf}_S \longrightarrow \mathit{Intf}_R$ is a function from the interface of $S$ to the interface of $R$. The function $\sigma$ maps every interface port of the operand $S$ to a corresponding interface port of $R$.

A transformation $M \overset{\sigma}{\rightsquigarrow} N$ is *based on* a structural operator $op$ if $op(Str(M)) = \langle R, \sigma\rangle$ such that $Str(N) \approx R$. When this property holds, we say that $M \overset{\sigma}{\rightsquigarrow} N$ *via op*. Note that $\approx$ is used instead of plain equality in this definition.

Next examples of some structural operators are given. All of these operators are intuitive. The formal definitions are omitted here; they have been included in a previous technical report [2]. It is possible to define the structural operators in a set-theoretic or in a graph grammar formalism. The set-theoretic approach is used here.

In what follows, each operator's syntactic form is specified as $op[P_1, \ldots, P_n]$, where $op$ is the name of the operator, and the $P_i$ are the secondary parameters involved (besides the operand). Thus the application of the operator to a bl-structure $S$ (the operand) is denoted by $op[P_1, \ldots, P_n](S)$.

## 4.1 Homomorphic Operations

Two structural operators, namely *substitution* and *extension* have particular importance because they induce bl-homomorphisms. When an arbitrary sequence of these fundamental operators are applied to a given bl-structure, the resulting bl-structure turns out to be homomorphic to the starting bl-structure. An example is provided in Figure 8. If the application sequence consists of only substitution operations, the induced bl-homomorphism is surjective. If it consists only of extension operations, then in the induced homomorphism the function $\gamma$ and the functions in $\Sigma$ are identity functions.

### 4.1.1 Substitution

$\mathit{Subst}[C_i, D, \delta]$

Substitution involves replacing a component of a given bl-structure with an instance of some other module. In order to apply this operator to a bl-structure $S$, the component $C_i \in \mathit{Cmps}_S$ to be replaced, the module $D$ (replace-

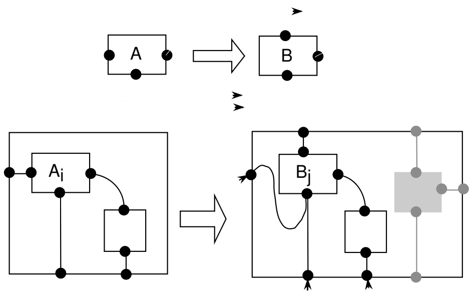

Figure 8: bl-homomorphism resulting from substitution followed by an extension. In the substitution, the replaced component is $A_i$ and the replacement module is $B$. The extension operation is indicated by the grey areas.

ment module) whose instance is to replace $C_i$ in $S$, and a function $\delta\colon \mathit{Intf}_C \longrightarrow \mathit{Intf}_D$ must be specified.

The result of the substitution is a new bl-structure in which $C_i$ is replaced by some instance $D_j$ of $D$ according to the interface mapping $\delta$ such that $D_j \notin \mathit{Cmps}_S$ (so that the new instance does not clash with other components of type $D$ in $S$.) The interface of the operand $S$ is mapped to the interface of the new module by the identity mapping on $\mathit{Intf}_S$. Substitution was illustrated in Figure 7.

In the bl-homomorphism induced by the substitution, $\sigma$ is the identity function on the operand, $\gamma$ is such that $\gamma(X_i) = X_i$ if $X_i \neq C_i$ and $\gamma(C_i) = D_j$ otherwise.

The substitution operator underlies the substitutivity principle. The substitutive closure of a transform relation can be defined in terms of this operator: a transform relation is closed under substitution (or with respect to the substitutivity property) if a module $N$ transforms to another module $M$ whenever the bl-structure of $N$ can be obtained from that of $M$ (up to $\approx$) through a sequence of substitutions such that the precondition $C \overset{\delta}{\rightsquigarrow} D$ is satisfied for each operation in the sequence.

### 4.1.2 Extension

$\mathit{Extend}[\mathit{Intf}^+, \mathit{Cmps}^+, \smile^+, \frown^+]$

Extension involves adding new interface ports, components, and connections to a given

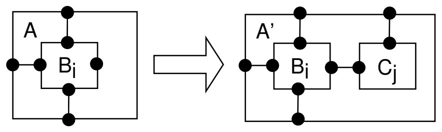

Figure 9: Extension.

bl-structure. In order to apply an extension operator to a bl-structure $S$, the sets of new interface ports $\mathit{Intf}^+$, components $\mathit{Cmps}^+$, internal connections $\smile^+$, and external components $\frown^+$ must be specified. The resulting bl-structure is obtained by taking the pairwise union of the old and new sets.

The new connections may involve the interface ports of both the old and the new components, as well as the ports of both the new and the old interface. For the operator to be applicable to an operand $S$, $\mathit{Intf}^+$, $\mathit{Cmps}^+$, $\smile^+$, and $\frown^+$ should be such that the resulting bl-structure is well defined.

As is the case for substitution, the interface of the operand $S$ is mapped to the interface of the extended module by the identity mapping on $\mathit{Intf}_S$. Extension is illustrated in Figure 9.

In the bl-homomorphism induced by the extension, all of the mappings involved equal the identity functions on the domains of those mappings. Extension is equivalent to structural inheritance without component exclusion [15].

## 4.2 Reversible Operations

Substitution and extension are not the only kinds of structural operations to be considered. It is possible to identify several other kinds of operators. Those discussed in this subsection should ideally preserve the external functionality of the modules to which they are applied, although this is not always guaranteed in every behavioral model.

All of these additional operators are reversible, in that if $M \overset{\sigma}{\rightsquigarrow} N$ via such an operator $op$, then $N \overset{\sigma}{\rightsquigarrow} M$ via an inverse operator $op^{-1}$. As in extension and substitution, the resulting function $\sigma$ is the identity mapping on the interface of the operand.

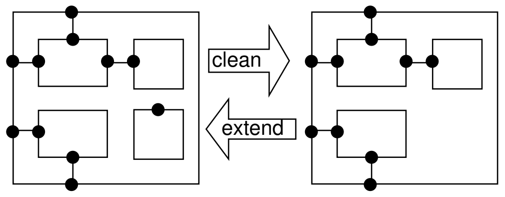

Figure 10: Clean operation.

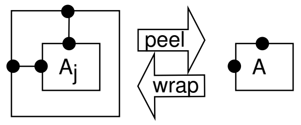

Figure 11: Wrap and peel operations.

### 4.2.1 Clean

*Clean*

The *clean* operator removes a non-essential component from a bl-structure, as shown in Figure 10. A component is essential if it is directly or indirectly connected to an interface port of the bl-structure. Formally, $C_i \in \mathit{Cmps}_S$ is an *essential component* of $S$ iff $p \frown_S q.C_i$ for some $q \in \mathit{Intf}_C, p \in \mathit{Intf}_S$, or $p.C_i \smile_S q.D_j$ for some $p \in \mathit{Intf}_C, D_j \in \mathit{Cmps}_S, q \in \mathit{Intf}_D$ such that $D_j$ is an essential component of $S$.

The inverse of a clean operation that removes component $C_i$ from a module is an extension operation which adds a loose component of the same type to the result of the clean operation.

### 4.2.2 Wrap and Peel

*Wrap*, *Peel*

The *wrap* operator encapsulates a bl-structure within another bl-structure as the sole component of the latter. Every interface port of the wrapped component is connected to a distinct interface port of the wrapper module.

The inverse of this operation is the *peel* operator, which extracts the wrapped component from a wrapper module by removing the latter's outer shell.

These two operators are illustrated in Figure 11.

### 4.2.3 Flatten and Cluster

*Flatten*$[C_i]$, *Cluster*$[X, M]$

The *flatten* operator unravels a bl-structure $S$ with respect to a given component $C_i$. In the resulting bl-structure, the component $C_i$ is replaced by $C$'s own components and connections. For every component $D_j$ of $C$, a new component $D_{j'}$ is added to $S$ such that the instance $D_{j'}$ does not clash with any previously existing instance of $D$ in $S$. The internal connections of the resulting bl-structure, say $R$, is set such that:

- if $p.D_j \smile_C q.E_k$ then $p.D_{j'} \smile_R q.E_{k'}$;
- if $p \frown_C q.D_j$ and $p.C_i \smile_S r.E_k$ then $q.D_{j'} \smile_R r.E_k$; and
- if $p \frown_C q.D_j$ and $r \frown_S p.C_i$ then $r \frown_R q.D_{j'}$.

This process corresponds to removing the outer shell of $C_i$ within $S$, thereby exposing its internal structure to $S$.

For a flatten operator to be applicable, $C$ must be a *non-leaf* module.

The inverse of this operator is called *cluster*. Clustering factors out a subset $X$ of a bl-structure's components by encapsulating $X$ within an instance of a proper module $M$. Then $M$ is called the abstraction module. The cluster and flatten operators are illustrated in Figure 12.

The cluster operation has the following precondition: *Str*$(M)$ must be $\approx$-equivalent to the bl-structure given rise by the subset $X$ (in the context of the enclosing bl-structure to which the cluster operator is applied). We require the abstraction module $M$ to be explicitly specified, although such a module can automatically be constructed from $X$.

# 5 Architectural Histories

It is possible to represent the architectural history of an evolving system as a DAG, as shown in Figure 3. We call such a graph an *evolution graph.*

Each node of an evolution graph denotes a *baseline* of the underlying system. A baseline is self-contained in that it does not contain references to objects outside it. As such it exists

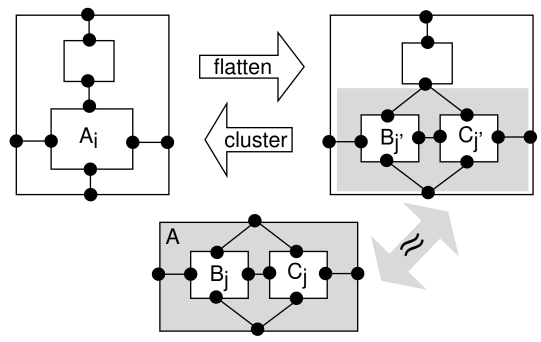

Figure 12: Flatten and cluster operations. The shaded inner box on the right represents the bl-structure associated with the subset $\{B_{j'}, C_{j'}\}$ of the enclosing bl-structure. $A$ is the underlying abstraction module.

independent of other baselines. Conceptually, a baseline represents a distinct version of the system at a given point in time, or as a variant of an ancestor baseline.

A branch in the graph defines an *evolution step.* Each evolution step maps a source baseline to a successor, and as such represents a *delta* of the source baseline.

The rest of this section demonstrates how the previously developed concepts can be used to specify this kind of system progression in terms of baselines and evolution steps.

## 5.1 Baselines and Evolution Steps

Suppose each baseline is specified in terms of a module system. With this in mind, let $\mathbf{M}_t$ denote the module system associated with the baseline $t$. A *path* in the evolution graph can then be represented as a sequence:

$$\mathbf{M}_0 \overset{1}{\Rightarrow} \mathbf{M}_1 \overset{2}{\Rightarrow} \cdots \overset{n}{\Rightarrow} \mathbf{M}_n$$

where $\overset{t}{\Rightarrow}$ represents the $t^{\text{th}}$ evolution step along the path. When step $\overset{t}{\Rightarrow}$ is applied to baseline $\mathbf{M}_{t-1}$, the successor baseline $\mathbf{M}_t$ is obtained.

By abuse of notation, we will use conventional set operators ($\cup$, $/$, $\subseteq$) on module systems. As a general rule, the set operator should be though of as being applied to the sets of modules of the operands.

An evolution step $\overset{t}{\Rightarrow}$ is defined in terms of:

- a module system $\mathbf{M}^+$ of *introduced* modules,
- a module system $\mathbf{M}^- \subseteq \mathbf{M}_{t-1}$ of *retired* modules, and
- a *base transform relation* $\rightsquigarrow$ over $(\mathbf{M}_{t-1}/\mathbf{M}^-) \times \mathbf{M}^+$, where $M \overset{\sigma}{\rightsquigarrow} M'$ means that module $M$ of baseline $t-1$ is transformed to module $M'$ in baseline $t$. Then module $M'$ is referred to as the next *version* of module $M$.

The baseline $\mathbf{M}_{t-1}$ is called the *source baseline*, and $\mathbf{M}_t$ is called the *target baseline*.

In addition, each evolution step $\overset{t}{\Rightarrow}$ must satisfy the following properties:

1. The next version of a module is unique when it exists: $M \overset{\sigma}{\rightsquigarrow} M'$ and $M \overset{\sigma}{\rightsquigarrow} M''$ implies $M' = M''$.
2. The target baseline includes all introduced modules and excludes all retired modules: $\mathbf{M}_t \supseteq (\mathbf{M}_{t-1} \cup \mathbf{M}^+)/\mathbf{M}^-$.

The base transform relation $\rightsquigarrow$ defines the modules which are directly updated by the evolution step. Only those changes that cannot be inferred through the substitutivity principle are included in this relation.

The set of introduced modules, $\mathbf{M}^+$, mainly contains the new versions of those modules in the source baseline $(\mathbf{M}_{t-1})$ that are to be updated by $\rightsquigarrow$ for inclusion in the target baseline $(\mathbf{M}_t)$. It may also contain brand new top-level modules to be introduced for the first time in the target baseline along the current path. The descendants (with respect to the dependence relation $<$) of these modules are also included for completeness purposes.

The set of retired modules, $\mathbf{M}^-$, contains those modules that are to be retired in the target baseline. For the target baseline to be well defined, none of its modules may depend on a retired module.

With these constraints, the substitutive closure of $\rightsquigarrow$ is a transform relation over $\mathbf{M}_{t-1} \times \mathbf{M}_t$. This relation determines the next version in the target baseline of a given module of the source baseline, provided the module changes version. Thus for $M \in \mathbf{M}_{t-1}$ and $N \in \mathbf{M}_t$, $M \overset{\sigma}{\rightsquigarrow}_* N$ means $M$ changes version from baseline $t-1$ to baseline $t$. In this case either $M \overset{\sigma}{\rightsquigarrow} N$ or $N$ is obtained from $M$ through a successive substitution of $M$'s components whose versions change, by the instances of the next versions of these components.

## 5.2 Versioning

A version number is attached to each module to keep track of its evolution along a given path in the evolution graph. A *versioned module* is denoted by $M^v$, where $v$ is the version number, and $M$ is the unique name of the module. Version numbers are independent of baseline indices, so that if $M^v \in \mathbf{M}_t$, it is often the case that $t \neq v$.

If a module does not change version from one baseline to the next, its version number remains the same. Otherwise, it is incremented. We assume that version numbers start at 0 for each new module name, and is incremented by one every time the module evolves into a new version.

Each baseline contains at most one version of a given module: $M^v, M^w \in \mathbf{M}_t$ implies $v = w$. Thus at step $t$, a module $M^v$ of baseline $t-1$ may evolve into a unique new version $M^{v+1}$ in baseline $t$.

In general, the system $\mathbf{M}^+$ associated with step $t$ may contain three types of modules:

a. $M^{v+1}$ where $M^v \in \mathbf{M}_{t-1}$: these are new modules not in the source baseline, but each represents a new version of some module in the source baseline. For each of these modules, the base transform relation associated with step $t$ must contain an entry $M^v \overset{\sigma}{\rightsquigarrow} M^{v+1}$.

b. $M^v$ where $M^v \in \mathbf{M}_{t-1}$: these are submodules of some module in $\mathbf{M}^+$, and do not appear in the domain of the base transform relation $\rightsquigarrow$. They are included in $\mathbf{M}^+$ to preserve well-definedness of $\mathbf{M}^+$.

c. $M^0$ where $(\forall v)(M^v \notin \mathbf{M}_{t-1})$: these modules are either new top-level modules, or they are descendants (submodules) of some module in $\mathbf{M}^+$. In either case, the name $M$ is introduced for the first time along the current path; hence the version number is 0. These modules do not appear in the base transform relation $\rightsquigarrow$ either.

### 5.3 Execution of an Evolution Step

Evolution steps act as *delta*s on baselines. Thus given a path in the evolution graph, baseline $t$ is obtained by executing the evolution step $t$ on baseline $t-1$. The execution algorithm guarantees that if $M$ changes version from baseline $t-1$ to $t$, then every module that depends on $M$ also changes version accordingly. Hence if $M^v \overset{\sigma}{\rightsquigarrow}_* M^{v+1}$ and $M^v < N^w$ ($M^v$ is a submodule of $N^w$) for some $N^w \in \mathbf{M}_{t-1}$, then $\mathbf{M}_t$ must contain a module $N^{w+1}$ such that $N^w \overset{\delta}{\rightsquigarrow}_* N^{w+1}$ for some $\delta$. If $N^w$ is not in the domain of $\rightsquigarrow$ then $\delta$ is the identity function on $\mathit{Intf}_{N^w}$; otherwise it equals $\beta$ where $N^w \overset{\beta}{\rightsquigarrow}_* N^{w+1}$. The module $N^{w+1}$ is the next version of $N^w$ in baseline $t$. It is obtained by substituting in $N^w$ an instance of $M^{v+1}$ for every instance of $M^v$.

The application algorithm first initializes $\mathbf{M}_t$ to $\mathbf{M}^+$, and then recursively adds the next versions of all affected modules in baseline $t-1$ that depend on some module in the domain of $\rightsquigarrow$. After this step, the remaining modules of baseline $t-1$ are copied to baseline $t$ provided that they are excluded from $\mathbf{M}^-$. These modules do not change versions in baseline $t$. Modules in $\mathbf{M}^-$ are omitted from baseline $t$.

If $M^v \overset{\sigma}{\rightsquigarrow} M^{v+1}$, $N^w \overset{\delta}{\rightsquigarrow} N^{w+1}$, and $M^v <^* N^w$, then an inconsistency arises. On the one hand, $N^{w+1}$ should be in $\mathbf{M}_t$ since it changes version by definition. On the other hand, the dependence of $N^w$ on $M^v$ ultimately gives rise to a new version of $N^w$ that may be different from $N^{w+1}$ of $\mathbf{M}^+$. However, the same baseline cannot contain both versions. The situation can be resolved by consolidating the two versions. Version consolidation is based on the substitutivity principle. First $N^{w+1}$ is initially excluded from baseline $t$. During initialization, $\mathit{Str}(N^w)$ is temporarily set to $\mathit{Str}(N^{w+1})$ in baseline $t-1$. Thus a module $M^{v+1} \in \mathbf{M}^+$ is initially included in $\mathbf{M}_t$ iff it does not depend on any other module in the domain of $\rightsquigarrow$. This ensures that if $N^{w+1}$ also depends on $M^v$, then $M^{v+1}$ replaces $M^v$ in the the consolidated, next version of $N$ to be included in the baseline $t$. In the end, $N^v \overset{\sigma}{\rightsquigarrow}_* N^{v+1}$ is always satisfied, as required. Figure 13 illustrates version consolidation.

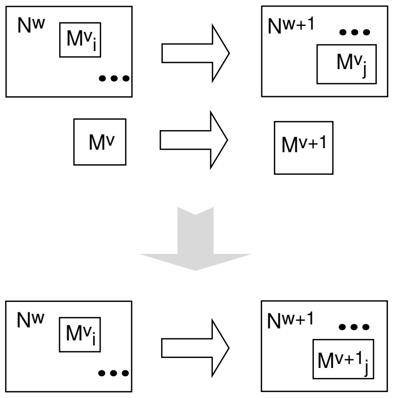

Figure 13: Version consolidation.

The execution algorithm with version consolidation is given as follows:

1. Initialization.

```
Set $\mathbf{M}_t$ to $\emptyset$;
for all ($N^w \in \mathbf{M}^+$) do
    if ($N^{w-1} \overset{\sigma}{\rightsquigarrow} N^w$ and $M^v <^* N^{w-1}$
     for some $M^v \overset{\delta}{\rightsquigarrow} M^{v+1}$) then
        set $Str(N^{w-1})$ to $Str(N^w)$
    else
        add $N^w$ to $\mathbf{M}_t$
```

2. Extension of $\mathbf{M}_t$ with modules that change version.

```
while ($\mathbf{M}_{t-1}$ contains a module that must
    change version in $\mathbf{M}_t$) do
```

   2.1 Pick a module $N^w$ from $\mathbf{M}_{t-1}$ such that $N^{w+1} \notin \mathbf{M}_t$ and $C^x < N^w$ for some $C^x \overset{\sigma}{\rightsquigarrow}_* C^{x+1}$. If such a module cannot be found, $\mathbf{M}_{t-1}$ does not contain any more modules that have to evolve to a new version in $\mathbf{M}_t$.

   2.2 Define a new module $N^{w+1}$. This is the next version of $N^w$, and is obtained from $N^w$ by simultaneously substituting for every component whose type changes version, by an instance of the next version of that component's type. Thus if $C^x \overset{\sigma}{\rightsquigarrow}_* C^{x+1}$ then $C^{x+1}$ is substituted for every component of type $C^x$ in $N^w$. A

```
    substitution operation is performed
    for all components that change version.
2.3 Add $N^{w+1}$ to $\mathbf{M}_t$.
```

3. Extension of $\mathbf{M}_t$ with modules that do not change version.

```
for all ($M^v \in \mathbf{M}_{t-1}$) do
    if ($M^v \notin \mathbf{M}^-$ and $M^{v+1} \notin \mathbf{M}_t$) then
        add $M^v$ to $\mathbf{M}_t$
```

Note that the algorithm does not address some inconsistencies that may arise due to retired modules. The conditions on $\rightsquigarrow$, $\mathbf{M}^+$, and $\mathbf{M}^-$ guarantee that no retired module changes version in the target baseline. Therefore, $M^v \in \mathbf{M}^-$ implies $M^{v+1} \notin \mathbf{M}^+$ is automatically satisfied. However, a consistency check is required to ensure that no module in $\mathbf{M}_t$ depends on a retired module after the execution of an evolution step. If this check is violated, then $\mathbf{M}_t$ is not well defined. Thus if $M^v$ is to be retired, all modules that depend on $M^v$, but do not change version must also be retired. If a module that depends on $M^v$ changes version, the next version of that module cannot depend on $M^v$. The applicable condition should be verified each time a module is included in $\mathbf{M}_t$.

## 5.4 Conceptual Operations

An evolution step may involve many discrete, simultaneous updates to a baseline. It is useful to identify these discrete, evolutionary updates as conceptual operations, and further decompose the evolution step into a sequence of such operations. Each conceptual operation can be expressed in terms of the involved modules, base transformations, mappings, sets, semantic operations, and constraints. This subsection defines some fundamental conceptual operations and relates them to familiar design concepts. A small concrete example was provided in Section 1, Figure 1.

### 5.4.1 Replacement

The most basic conceptual operation we identify is *replacement*. It involves replacing an arbitrary module $N^w$ in baseline $t-1$ by an arbitrary new module $N^{w+1}$ in baseline $t$. A replacement operation can be expressed as follows:

- *Modules*
  - $N^w \in \mathbf{M}_{t-1}$: module to be replaced
  - $N^{w+1} \in \mathbf{M}^+$: replacement module, next version of $N^w$
- *Mappings and Sets*
  - $\sigma\colon \mathit{Intf}_{N^w} \longrightarrow \mathit{Intf}_{N^{w+1}}$: replacement mapping
- *Transformations*
  - $N^w \overset{\sigma}{\rightsquigarrow} N^{w+1}$.

Here the transformation $N^w \overset{\sigma}{\rightsquigarrow} N^{w+1}$ is not based on a particular structural operation. Its soundness must be justified by some external means.

Replacement is a global operation, in that all the modules that depend on the replaced module will be affected, and as such, these modules will evolve into their respective new versions in the target baseline.

### 5.4.2 Selective Replacement

If the effect of replacing a module by another is to be confined to a particular context, then the corresponding conceptual operation is called *selective replacement*. Here instead of an entire module $N^w$, a selected component of $N^w$ is replaced. Thus $N^w$ specifies the context in which the replacement is applicable. The operation affects only $N^w$ and the modules which depend on $N^w$. It does not affect the replaced component or the modules which depend on it. The underlying transformation is based on the substitution operator.

- *Modules*
  - $N^w \in \mathbf{M}_{t-1}$: context, or module whose component is to be replaced
  - $C_i^x \in \mathit{Cmps}_{N^w}$: component of $N^w$ to be replaced
  - $D^y \in \mathbf{M}^+$: replacement module whose instance is to replace $C_i^x$ in the next version of the context

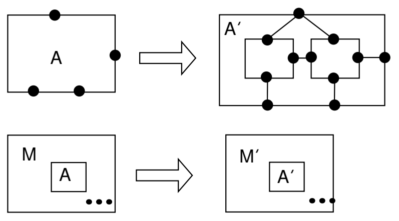

Figure 14: Structural refinement of $A$ to $A'$. $A$ is the replaced module and $A'$ is the replacement module.

  - $N^{w+1} \in \mathbf{M}^+$: next version of context after the replacement

- *Mappings and Sets*
  - $\sigma\colon \mathit{Intf}_{C^x} \longrightarrow \mathit{Intf}_{D^y}$: replacement mapping

- *Transformations*
  - $N^w \overset{id}{\rightsquigarrow} N^{w+1}$ via $\mathit{Subst}[C_i^x, D^y, \sigma]$.

### 5.4.3 Structural Refinement

A replacement operation is called *structural refinement* when the replaced module ($N^w$) is a leaf module and the replacement module ($N^{w+1}$) is a non-leaf module. This is illustrated in Figure 14.

Note that the internal structure of a leaf module is undefined. Structurally refining such a module can be thought of as defining its internal structure in terms of the composition of instances of other modules.

Since structural refinement is a special case of replacement, we need only to add some constraints to the generic specification of the replacement operation.

- ...

- *Constraints*
  - $N^w$ is a leaf module
  - $N^{w+1}$ is a non-leaf module
  - $\mathit{Intf}_{N^w} = \mathit{Intf}_{N^{w+1}}$
  - $\sigma$ is the identity function.

### 5.4.4 Abstraction

*Abstraction* supports reuse. It is applicable upon the recognition that a certain subset of design elements is best treated as a single unit. The structural operator underlying abstraction is the cluster operator.

Abstraction involves first the identification of the subset of components to be abstracted in a given module (or set of modules). Once this subset is identified, it can be factored out and encapsulated in a separate module, whose instances then replace the abstracted subset in all affected modules. A module is affected by an abstraction, if as a result of that abstraction, the module has to evolve to a new version in the subsequent baseline.

- *Modules*
  - $\mathbf{K} \subseteq \mathbf{M}_{t-1}$: modules affected by the abstraction (a module is affected by an abstraction if a cluster operation is to be applied to it)
  - $A^0 \in \mathbf{M}^+$: the abstraction module
  - $K^{v+1} \in \mathbf{M}^+$ for every $K^v \in \mathbf{K}$: new versions of the modules affected by the abstraction

- *Mappings and Sets*
  - $X_K$ for every $K^v \in \mathbf{K}$: for each module $K^v \in \mathbf{K}$, a subset $X_K$ of $K^v$'s components to be factored out from $K^v$

- *Transformations*
  - $K^v \overset{id}{\rightsquigarrow} K^{v+1}$ via $\mathit{Cluster}[X_K, A^0]$, for every $K^v \in \mathbf{K}$

- *Constraints*
  - $A^0 \notin \mathbf{M}_{t-1}$
  - $\mathit{Str}(A^0) \approx \mathit{Str}(X_K)$ for every $K^v \in \mathbf{K}$, where $\mathit{Str}(X_K)$ is the bl-structure associated with the subset $X_K$ in context $K^v$.

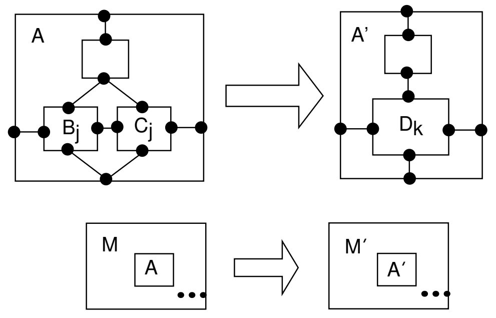

Figure 15: Abstraction of $A$ into $A'$. Here $A \in \mathbf{K}$, $X_A = \{B_j, C_j\}$, and $D$ is the abstraction module.

### 5.4.5 Extension

As the name suggests, this conceptual operation is based on the extension operator illustrated in Figure 9. Its specification is thus straightforward.

- *Modules*
  - $N^w \in \mathbf{M}_{t-1}$: module to be extended
  - $N^{w+1} \in \mathbf{M}^+$: the extended module
- *Mappings and Sets*
  - $\mathit{Intf}^+, \mathit{Cmps}^+, \smile^+, \frown^+$: new interface ports, components, and connections to be added to $N^w$
- *Transformations*
  - $N^w \overset{id}{\rightsquigarrow} N^{w+1}$ via $\mathit{Extend}[\mathit{Intf}^+, \mathit{Cmps}^+, \smile^+, \frown^+]$
- *Constraints*
  - usual preconditions on $\mathit{Intf}^+$, $\mathit{Cmps}^+$, $\smile^+$, $\frown^+$ so that $\mathit{Extend}[\mathit{Intf}^+, \mathit{Cmps}^+, \smile^+, \frown^+]$ is applicable to $N^w$

## 6 Summary and Discussion

This paper presented a formal framework for box-and-line type structure diagrams commonly used in software architecture and system modeling techniques. It demonstrated how the framework can be used to record the architectural history of an evolving system in terms of structural transformations. At each point in the evolution graph of a system, the configuration of the system at that point is represented by a set of interdependent modules, called a baseline. An evolution step represents a *delta* of the system architecture, and as such, maps a baseline of the graph to its successor by specifying a set of structural transformations. These transformations can be based on formal operators on box-and-line structures. Several such operators were defined to represent some familiar design concepts such as abstraction, structural refinement, and extension (or structural inheritance). The evolution of the system satisfies the fundamental property of substitutivity, which states that if a module evolves by a structural transformation from one baseline to another, then each module that depends on that module will also simultaneously evolve by a respective structural transformation.

To support the formal model, an equivalence relation, $\approx$, was defined on box-and-line structures. Two structures are equated according to this relation if their top-level graphs are isomorphic. This gives rise to a semantic framework in which the equivalence relation could play the role of equality. Although the relation was sufficient for the purposes of this article, it is not preserved by most structural operators, including substitution (more precisely, it is not a congruence with respect to most structural operators). This casts doubt to its suitability as a semantic relation. To overcome this problem, a weaker relation based on nested isomorphism can be adopted, so that two structures are equated not only when their top-level graphs are isomorphic, but also when the graphs of their components are pairwise isomorphic, down to the bottommost layer. It is well known that such relations can be defined succinctly as a least fixpoint of some recursive relational property (several examples can be found in the literature on semantics of concurrent programs [11].) In our case, the sought weaker semantic relation would be the smallest relation $\mathcal{R}$ which satisfies the following recursive property (or the least fixpoint of the following relational property on $\mathcal{R}$):

$S \ \mathcal{R} \ R$ iff there exists a bl-isomorphism $\langle \sigma, \gamma, \Sigma \rangle$ from $S$ to $R$ such that $\mathit{Str}(C) \ \mathcal{R} \ \mathit{Str}(\mathit{Typ}(\gamma(C_i)))$ for every $C_i \in \mathit{Cmps}_S$.

Note that this weaker relation does preserve all the structural operators defined, as desired.

Since intuitively two structurally equivalent systems must exhibit similar external functionalities or black-box behaviors, structural equivalence should subsume functional or behavioral equivalence. Note however that here the structural framework assumes nothing about the particular behavioral model in which the components are assigned their respective external functionalities. In structural transformations, homomorphisms, and operators, the semantic mappings used only state a containment relationship based on the local semantic roles played by individual interface ports. This relationship is meaningful in reasoning about the structural evolution of a system, but is not necessarily meaningful for reasoning about its external functionality. The external functionality of a system is often more than the sum of its interface ports, and therefore, it is not sufficient to consider the semantic role of each interface port in isolation. The external functionality usually also takes into account the interdependencies between the the individual interface ports. The semantic properties of the composition constructs of the behavioral model dictate these interdependencies. This view underlies many semantic theories of concurrent programs, where the relation between system structure and system behavior (external functionality) has been studied extensively [6, 10]. For example, whereas extension in general does not preserve external functionality, substitution, in most behavioral models, preserves the external functionality of its operand under the assumption that the external functionality of the replacement module $D$ subsumes that of the replaced module $C$.

The notion of baseline originates from software configuration management, which addresses the management of components in an evolving software project [5]. Unlike traditional configuration management, here we do not address evolution at the source code level, but rather at the architectural level. Since there is no conceptual difference between versions that are variants (or specializations) of a common parent system and versions that are temporal successors of a parent system, both notions of version are captured simultaneously in the evolution model.

The evolution model presented here is best applicable to the development of concurrent real-time distributed software in the context of such specification techniques as ROOM [15], SDL [14], or DSL [19], where the runtime architecture of a system can conveniently be represented in terms of box-and-line diagrams. Luckham et al. refer to such architectures as interface connection architectures [9]. Hence the evolution model is equivalently suitable for use with architecture description languages [1, 3, 16, 4], which adopt this architectural paradigm.

However the evolution model presented has its limitations. For example, to maintain some degree of external consistency, refining interface ports (interface refinement) is not allowed. In other words, it is illegal to split an interface port of a module to several interface ports in a subsequent version. This is a serious limitation, since such refinement is not uncommon in real systems: it often happens that a particular interface eventually becomes too overloaded, at which point it may be desirable to split that interface in the next version of the system. Interface refinement is difficult to express as a homomorphic operation or as a *delta* on an existing system because it is context-dependent. If an interface port $p$ of a module $M$ is split into two ports $p_1$ and $p_2$ in a subsequent version, some connections that were originally bound to $p$ will be bound to $p_1$, while others will be bound to $p_2$. To compound the problem, the binding pattern is determined by the context (the dependent modules) when several modules depend on $M$. In such cases, a simple function on interface ports would not be sufficient to specify the resulting, complex semantic correspondences. The problem is deeper than a simple representation issue, and its treatment is considered as future work.

## About the Author

Hakan Erdogmus is a research officer with NRC's Institute for Information Technology. He joined IIT's Software Engineering Group in 1995. Mr. Erdogmus received a doctoral degree in Telecommunications from Université du Québec, Institut national de la recherche scientifique in 1994, and a Master's degree in Computer Science from McGill University in 1989. Prior to joining NRC, he was a research associate at INRS-Télécommunications. His research interests include formal methods for software engineering, software architecture, modeling and analysis of concurrent systems, and software engineering economics.

## References

[1] R. Allen and D. Garlan. Beyond definition/use: Architectural interconnection. In *Proc. of Workshop on Interface Definition Languages*, January 1994.

[2] H. Erdogmus. A formal framework for software architectures. Technical Report ERB-1047, National Research Council of Canada, Institute for Information Technology, Ottawa, Ontario, December 1995.

[3] D. Garlan, R. Allen, and J. Ockerbloom. Exploiting style in architectural design environments. In *SIGSOFT'94, Proc. 2nd ACM SIGSOFT Symp. on Foundations of Software Engineering*, pages 175–188, December 1994.

[4] D. Garlan and R. Monroe. Acme: an architecture description interchange language. In *Proc. CASCON'97, 7th Annual IBM Centre for Advanced Studies Conference*, Toronto, Ontario, November 1997.

[5] M. Genteman et al. Commercial real-time software needs different configuration management. In *Proc. 2nd Intl. Workshop on Software Configuration Management*, Princeton, NJ, October 1989.

[6] J. Hinterplattner, H. Nirschl, and H. Saria. Process topology diagrams. In *Proc. 3rd Internat. Conf. on Formal Description Techniques*, pages 535–550, 1990.

[7] F. Jananian and A. Mok. Modechart: A specification language for real-time systems. *IEEE Trans. Softw. Eng.*, 21(12), December 1994.

[8] D. C. Luckham. Rapide: A language and tool set for simulation of distributed systems by partial orderings. In *Proc. DIMACS Partial Orders Workshop IV*, Princeton University, N.J., July 1995.

[9] D. C. Luckham, J. Vera, and S. Meldal. Three concepts of architecture. Technical report, The Program Analysis and Verification Group, Computer Science Department, Stanford University, Stanford, CA, July 1995.

[10] G. Milne and R. Milner. Concurrent processes and their syntax. *J. Assoc. Comput. Mach.*, 26(2), April 1979.

[11] R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[12] D. E. Perry. Software interconnection models. In *Proc. ICSE'87, 9th Internat. Conf. on Software Engineering.* IEEE Computer Society Press, March 1987.

[13] Rational Software Corporation. UML Notation Guide, version 1.1. Available at *www.rational.com/uml*, September 1997.

[14] A. Sarma. Introduction to SDL-92. *Comput. Netw. ISDN Syst.*, 28(12):1603–1615, June 1996.

[15] B. Selic, G. Gullekson, and P. T. Ward. *Real-Time Object-Oriented Modeling.* Wiley, 1994.

[16] M. Shaw. Abstractions for software architecture and tools to support them. *IEEE Trans. Softw. Eng.*, 21(6), April 1995.

[17] M. Shaw and D. Garlan. Characteristics of higher-level languages for software architecture. Technical Report CMU-CS-94-210, Carnegie Melon University, School of Computer Science, Pittsburgh, PA, December 1994.

[18] Sun Microsystems, Inc. JavaBeans 1.01 Specification. Available at *www.java.sun.com/beans/docs*, July 1997.

[19] O. Tanir. *Modeling Complex Computer and Communication Systems: A Domain-Oriented Design Framework.* McGraw Hill, 1996.