A Formalism for Conformance Analysis and Its Applications

Tien N. Nguyen and Ethan V. Munson
Department of Computer Science
University of Wisconsin-Milwaukee
{tien,munson}@cs.uwm.edu

Abstract

Software development is a dynamic process where engineers constantly refine their systems. As a consequence, all software artifacts and their logical relationships evolve. At times, the evolutionary changes may reduce the consistency of the software project and break semantic connections among documents. We use the term conformance *to denote the state where the network formed by software documents and their relationships is in semantic harmony.* Conformance analysis *is the process of determining whether software documents and their logical relationships are in agreement. In our previous research, we have developed a formalism that can be used to verify strategies to conformance analysis. In this paper, we describe modifications to that formalism, and its applications to build conformance analysis tools in an integrated development environment that is extensible to incorporate new conformance analysis strategies and to combine multiple strategies together.*

1 Introduction

Software development is a dynamic process where engineers constantly modify and refine their systems. As a consequence, the software artifacts and their logical relationships always evolve over time. One software document may be logically related to another, which in turn is involved in relationships with one or more other documents. Therefore, documents and their logical relationships can be seen as a network. The state of the network in which a project's software documents are in perfect semantic harmony with each other and with the semantics of their relationships is referred to as *conformance*. The evolution of these documents may result in the breaking of the conformance state of the project and of the semantic connections among documents as well. We use the term *conformance analysis* to refer to the process of determining whether software documents and their logical relationships are in agreement.

Conformance analysis tools to either automatically detect non-conformance or at least automate the detection of possible non-conformance among software documents are needed. In the latter case, automated detection can be used to guide developers to likely problem areas. In general, the level of automatic detection of different conformance analysis strategies depends on the nature of the document contents. The software documents can be broadly divided into two categories: *formal* and *informal*. Formal documents are often written in some formal or programming languages, which can have precisely defined semantics. Therefore, formal documents can be understood and analyzed by computerized tools such as compilers. Examples are source code, UML specifications, Z specifications, Petri Nets, etc. Informal documents include many other documents, for example, requirements written in natural languages, and multimedia design documentation. A software artifact can also be a combination of both categories.

Strategies to perform conformance analysis might be different for various kinds of software documents. Conformance analysis strategies can be broadly divided into two categories: *semantics-dependent* and *semantics-independent*. A strategy that is required to have the knowledge of the semantics of documents is referred to as *semantics-dependent*. This type of analysis strategy is usually applied to formal documents. For example, it may be possible to automatically derive information about dependencies between C files based on facts about C bindings [19]. This strategy has been applied in the area of build optimizations, where the semantics of the underlying items may be very helpful in avoiding a costly build process [19]. This type of strategy can also be seen in approaches that formalize documentations using natural language processing or knowledge engineering techniques [10].

Semantics-independent strategies do not require any knowledge about document semantics. Among them, the *make* [6] strategy for the build process is a *timestamp strategy* that examines the timestamps of source files and the files that they generate. That is, if the most recent modification timestamps of source files are older than those of the target files, then the targets do not need to be rebuilt.

In contrast, the *difference strategy* is very time-consuming since it compares the content of the old and new versions of a document in order to decide whether non-conformance might have occurred. To avoid this cost, an approximation approach is often used such as the *signature* method, where a document is not considered changed unless its signature (i.e. some document property) is changed. More conservative strategies are those analyses that require *human intervention*. This type of strategy attempts to automate the detection of possible non-conformance, guiding developers to potential problems.

To explore solutions to the conformance problem, we have developed a formalism [12] that formally describes the conformance analysis process and strategies. This paper is focused on modifications and applications of that formalism to build conformance analysis tools in an environment that is extensible to incorporate new conformance analysis strategies and combine strategies together. The next section discusses related work. Section 3 describes our representations of software documents and logical relationships in a manner suitable for conformance analysis in the Software Concordance (SC) environment. Section 4 presents basic elements of our conformance model and the formalism is in Section 5. Section 6 describes conformance analysis tools in SC. Our conclusions appear in the final section.

2 Related work

Inconsistency management has been well studied [14, 17]. According to Spanoukadis and Zisman [17], inconsistency management can be viewed as a process composed of six activities: detection of overlaps between software artifacts, detection of inconsistencies, diagnosis of inconsistencies, handling of inconsistencies, history tracking of inconsistency management process, and specification of an inconsistency management policy. The activities to be taken depend on the type of inconsistency being addressed. The methods developed to support these activities are based on logics [8, 20], model checking [1], formal frameworks [16], human-centered approaches [20], explicit policies [5], knowledge engineering [10], hypertext [9], abstract dependence graph [2], static checking [11], etc.

The problem with many existing approaches to traceability and inconsistency management is that they are effective for only a limited portion of the development process (called *within-model* management approach), while having little support for software product fragments from other parts of the software life cycle. Traceability and inconsistency management support is most frequently found between representations such as formal specifications and source code that are amenable to automated analysis. Little support exists for managing relationships between these representations and less formal representations. Our research addresses these problems via a *model-to-model* approach where a mechanism is provided to incorporate different methods to handle different types of artifacts.

During a software development process, changes in source artifacts may potentially break the dependency relationships and cause the software system out-of-date. Automatic system build tools, such as *make* [6], have to ensure that changes in source artifacts are properly reflected in dependent artifacts. The software build task can be seen as a form of conformance analysis where only the dependency relationships that automatically create derived artifacts are considered. Gunter [7] has developed a general theory of build optimizations based on a theory of concurrent computation over a class of Petri Nets called *p-nets*. Formal techniques in Gunter's model are applied into our formalism as will be described later.

3 Software documents and relationships

3.1 Software documents

Software documents and their logical relationships can be modeled as a network of nodes and links where each node represents a fragment of a document and each link represents a relationship between fragments. To model software documents, SC follows a *structure-oriented* approach where each document is considered to be logically structured into fine units, called *structural units* or *logical units*. Each software document is represented by a *tree* or a *graph* in which each node encodes a logical unit. This approach is often taken in *structured document* research, e.g. SGML and XML. Since XML has become the standard structured document format and very successful in representing many different data types, it is very natural to use XML for representing non-program artifacts. For a program, abstract syntax tree (AST) perfectly represents its logical structure. The structure-oriented representation also allows for the intermixture of different document types within a document. Via this approach, we have a uniform structure-oriented representation for many types of software artifacts [13].

To provide supports for different types of software documents, we have built the Software Concordance (SC), an integration of structured document editors for XML, HTML, and text documents, a syntax-recognizing Java program editor, an SVG graphic and animation editor, and a UML editor. SC is compatible with XML-based document editing environments since it supports the integration of new editors for new document types whose internal representations are XML-compatible. Integration requires only that a new editor follows a simple plug-in protocol. SC uses a Document Object Model (DOM) parser [3] to import XML-based documents, converts the DOM trees into SC's internal representation. For example, the Thorn UML editor [18] and

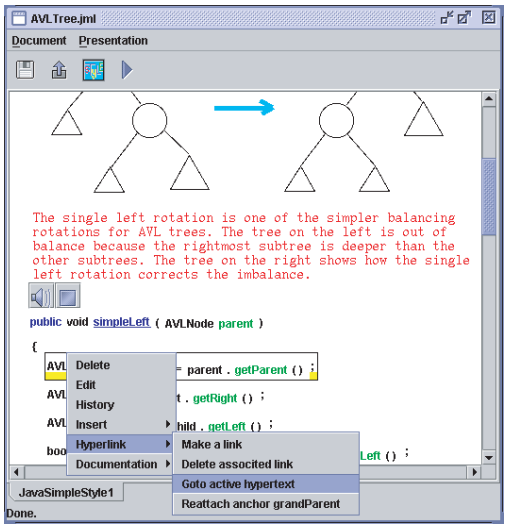

Figure 1. Document editor

the DrawSWF SVG editor [4] were easily integrated into the SC environment because their document representations are XML-based. Figure 1 shows a snapshot from the Java structured editor. To display a document, a default CSS-like style sheet is selected by the system for the document unless it has one. A user can choose to open a document with any appropriate style sheet.

To edit a document, the user moves the mouse and selects any structural unit of the document that needs to be edited. Then, via the commands in the pop up menu, the user can choose to edit the content of that structural unit presented in the selected portion of the presentation. The SC editor invokes the *node editor*, which is a simple ASCII text editor. The editor unparses the node and displays the resulting textual representation of the node to be edited. The user edits the text and returns to the document window. Depending on the type of the document that is being edited, the editor invokes either an XML, HTML parser or a Java parser to *incrementally* parse the modified text, and then creates new nodes and attaches them to the document tree. If there exists any errors in the modified text, error messages are displayed and the user can fix them. SC also allows a user to edit image or graphic documentation, and then to associate them with any structural unit in a Java program.

3.2 Document relationship networks

Because document relationships are involved in the domain of linked documents, it is natural to use the *hypertext model* as a basis for representing these relationships. The hypertext model is defined as a set of intellectual *works* and their inter- and intra-work relationships, represented by *links*. A *work*, representing for a software document, is an artifact that can be drawn from any medium, such as text, image, or video. A *link* (or *hyperlink*), representing for a logical relationship, is a first-class entity and defined as an association among a set of works or *anchors*. *Anchors* denote regions of interest within works by *referring* to a structural unit. Anchors form the endpoints for links. A link can connect multiple anchors and an anchor can belong to multiple links. Differing from *embedded* HTML anchors and hyperlinks, hyperlink structure of anchors and links in SC is separated from document contents, allowing multiple hyperlink structures on the same set of documents without modifying document contents. Links and anchors can be associated with any attribute-value pairs. Links may have named types (such as "requires," "must agree," or "motivates"), but the set of types is not fixed, so that different software processes can be supported.

In Software Concordance, logical relationships are divided into *causal* and *non-causal* classes. Causal relationships carry with them an implied logical ordering of the documents involved. For example, testing and bug reports cannot be produced until an implementation is available, and while it is not necessarily the case that requirements will be written before designs, there is certainly a logical relationship between them that makes design depend on requirements. A causal relationship can be considered as a *relation* between entities: something happens and causes something else to happen. Each *causal link* represents a causal relation, R. Causal links are always directional, connecting a set of *source anchors* and a set of *target anchors*. The source and target sets of a causal link l are denoted by S_l and T_l , respectively. If a link's source and target sets each have one element, the link is binary. *Non-causal links* exist when documents or parts of them must agree with each other, but the causality cannot be clearly identified. Each non-causal link represents a non-causal relation, designated by the symbol r. Non-causal links are logically non-directional. Let us use A_l to denote the set of anchor nodes connected by a non-causal link l.

A set of links and connected software documents form a *conformance graph*. Formally, the conformance graph is a five-tuple

$$G = \{N, L_C, L_N, E_D, E_N\}$$

where N is a finite set of anchor nodes, L_C is a finite set of causal link nodes, L_N is a finite set of non-causal link nodes, E_D is a finite set of directed edges and E_N is a finite set of non-directed edges. Each directed edge in E_D either connects a source anchor to a causal link or a causal link to a target anchor. Each non-directed edge in E_N connects an anchor to a non-causal link. Let $L = L_C \cup L_N$. Note that each relationship is represented by a node, rather than by an edge, as is often the case in other graph representations of dependencies. This creates more flexibility in representing logical relationships among multiple software documents.

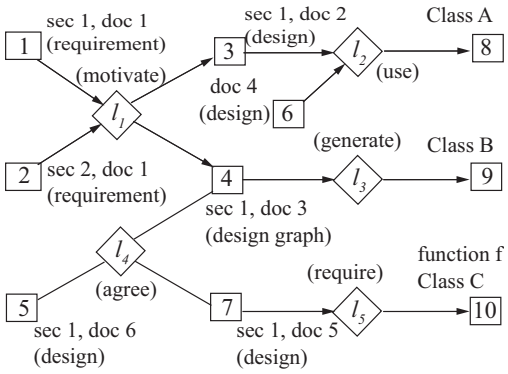

Figure 2. A conformance graph

Figure 2 shows an example of a conformance graph. There are ten document nodes in N. The links l_1, l_2, l_3, l_5 are causal links, which are all in L_C. L_N contains only l_4. Causal links have incoming and outgoing edges that are directed. Non-causal links have non-directed edges. The conformance graph must be acyclic with respect to causal links, because cycles would imply that a node causes itself. An anchor cannot be a target of more than one link. Conformance links may not be degenerate: 1) Non-causal links must connect at least two anchors, and 2) Causal links must have non-empty source and target sets. The SC environment provides editing supports for conformance graphs including anchor and link services. Anchor services include deleting an anchor, adding an anchor into the active link, removing an anchor off some link, renaming an anchor, and displaying the structural unit that the anchor refers to. Services for links include link creation, deletion, renaming, attribute's value viewing, and link history viewing. From the SC editor, users can define an anchor on any structural unit (see Figure 1) and add it into a conformance graph.

4 Conformance model

This section describes our approach in modeling the conformance analysis process and strategies. The conformance graph, consisting of links and related anchor nodes, is referred to as a *hypertext network*. The model is based on three key concepts: *interpretations*, *states*, and *operations*.

An interpretation is a way of viewing a hypertext network. For our purposes, there are two kinds of interpretations of a hypertext of documents and relationships.

- The *standard interpretation* is a distinguished way of viewing the hypertext network in which an anchor node (node, for short) represents the contents of a structural unit of a software document, and a link represents the semantics of the relationship between the nodes that it connects. The standard interpretation is a formal notion designed to represent the natural semantics of software documents. There is only one standard interpretation.

- An *abstract interpretation* is any alternative view of the hypertext network. For conformance analysis, there will be at least one abstraction that provides information useful to the conformance analysis process, which is called a *conformance analysis abstract interpretation*. Since we discuss conformance analysis, let us call it an *abstraction* for short. An abstraction is designated to represent a conformance strategy.

A *state* is a binding of some values in some domains to nodes and links. A state in the standard interpretation assigns the contents of fragments in documents to the corresponding nodes and the semantics of relationships to links. A state in an abstraction is called an *abstract state*, which assigns some values (called *abstract values*) to nodes and links for the purpose of detecting conformance problems.

An *operation* is an action by some agent that changes the states of the interpretations. A special *fix* operation of the standard interpretation refers to modifications made to the hypertext network in order to make documents conformant to each other with respect to a relationship. For each abstraction, there is a special *update* operation which will change abstraction values of the hypertext network for the purpose of tracking possible conformance problems. When a node's content is changed, the current states in the standard interpretation and abstractions also change.

An example of an abstraction is a timestamp abstraction. For instance, a timestamp abstraction contains assignments of timestamps to nodes and links in the hypertext network. Each anchor node has a modification timestamp and a link has a validation timestamp that records the time when the link was last validated by human inspection. The timestamp abstraction has an abstract state that is an assignment of the latest modification timestamps to the nodes and of latest validation timestamps to links. For causal links, four timestamps are relevant: $t_{src_{max}}$, the maximum timestamp of the source anchors; $t_{dest_{min}}$, the minimum timestamp of the destination anchors; $t_{dest_{max}}$, the maximum timestamp of the destination anchors; and t_{valid}, the validation timestamp of the link. If $t_{src_{max}} \leq t_{dest_{min}}$ and $t_{dest_{max}} \leq t_{valid}$ then the destinations are newer than the sources and the relationship was validated by a human sometime after the last changes to both. So, the conformance of the link and related nodes is reached. In the case of a non-causal link, assume that the non-causal link connects multiple anchors and let t_i be the timestamp of the i^{th} anchor. Then, if there does not exist any t_i such that $t_i > t_{valid}$ (i.e. nothing changes since the last validation), then the link is consistent.

Let us revisit the example of a conformance graph in Figure 2. Let us denote the timestamp for any node i by t_i,

and the timestamp for any link l_j by t_{l_j}. Suppose that the graph has no conformance problems. This means that for all causal links $t_{src_{max}} \leq t_{dest_{min}}$ and $t_{dest_{max}} \leq t_{valid}$, and for the non-causal link, l_4, $t_i \leq t_{l_4}$ for all nodes i connected by l_4. Suppose that now, section 1 of document 5 is changed. In this case, the abstract value of node 7 is also changed by the update operation ($t_7^{'}$) to reflect the most recent modification timestamp. Now, $t_7^{'}$ is greater than t_4, t_5, and t_{10}. If a conformance analysis is requested, the abstract values (timestamps) of related nodes are examined. According to the timestamp abstraction, the fact that the source of the causal relationship l_5 (node 7) is newer than its destination (node 10) is an indication that node 7 and node 10 will not be in agreement with each other with respect to the semantics of the link l_5. That is, the design choice in section 1 of document 5 might not require the use of the function f of the Class C anymore. Similarly, nodes 4,5, and 7 might not be in agreement anymore.

These points of possible non-conformance will be brought to the developers' attention. Suppose that after examining the contents of nodes, developers make some changes to documents to regain the conformance. These changes cause the change of the standard state. This action is modeled by the ***fix*** operation. Note that the fix operation can be done by any agent, which could be an automatic tool. Then, after the fix operation is invoked, the *update* operation is also invoked to update timestamps for nodes and links in the hypertext network.

5 Formalism

This section presents our conformance formalism. More details can found in [12]. First of all, we need to define some important concepts. Remember that N and L are the sets of anchor nodes and links respectively, and R and r are a causal relation and a non-causal relation, respectively. S_l and T_l are the source and target sets of anchors of a causal link l. A_l is the set of anchors of a non-causal link l.

5.1 Important concepts

The transitive and reflexive closure of edges corresponding to causal links defines a partial order set on a subset of nodes. The *minimal* node elements of this partial order set are called *source nodes*.

Left-closed: A subset X of $N \cup L$ is *left-closed* when $x \in X$ and either $R(y, x)$ or $r(y, x)$, then $y \in X$. In other words, a subset X of the hypertext network is left-closed if for every node x in X, any nodes related to x by a non-causal relationship directly or indirectly, and any nodes that are the causes of x directly or indirectly, also belong to X. Note that a left-closed subset does not have any incoming arcs whose source is outside the set.

Target-closed: A subset X of $N \cup L$ is *target-closed* if, for every causal link $l \in L$, $T_l \cap X \neq \emptyset$ implies $T_l \subseteq X$. That is, X is target-closed when for every causal link l in X, X either contains all of target nodes of l or none of them.

Marking: A *marking* M on a hypertext network is a left-closed and target-closed subset of $N \cup L$. The restriction that M is left-closed and target-closed means that for every causal link l, M either contains all of its target nodes or none of them (target-closed) and that M has no incoming links (left-closed). The idea behind the definition of a target-closed subset is that during the conformance analysis process, a causal link either has not been examined or has been examined at the same time with all of its targets.

5.2 States

For a node $n \in N$, V_n is defined as the domain of the values assigned to the node n. For the standard interpretation, V_n is the set of strings representing the content of the structural unit corresponding to the anchor node n. For abstractions, V_n is the domain of abstract values. These values are used to detect non-conformance.

For a link $l \in L$, in the standard interpretation, the value of l is the relation between the values of anchor nodes of the link l. In an abstraction, for a link l, let v_l be the abstract value of the link l in some domain V_l, which is also used to detect non-conformance. For each link l, the *constraint relation* of l, c_l, is defined as a relation between the abstract values assigned to anchor nodes of the link l and the abstract value assigned to l, v_l. An abstraction uses c_l to detect conformance problems. A *state* in an interpretation is the collection of all values assigned to *nodes* and *links*.

5.3 Consistent links

In the standard interpretation, a link l is said to be *consistent* in a state s if the nodes connected by l are conformant to each other with respect to the natural semantics of l. For abstractions, a link l is said to be *consistent* in state s of an abstraction B if and only if the constraint relation c_l holds. It is denoted by the consistency predicate $s \vdash_B l$. In other words, state s in the context of an abstraction B entails the consistency of the link l.

For example, let us consider the timestamp abstraction B for causal links. $V_n = \{0, 1, 2, 3, ...\}$ for all nodes n. $V_l = \{0, 1, 2, 3, ...\}$ for all links l. For each link l and the state t of B, $t \vdash_B l$ holds if $\max\{t_n \mid n \in S_l\} \leq \min\{t_m \mid m \in T_l\}$ and $\max\{t_m \mid m \in T_l\} \leq t_l$. That is, a relationship is considered consistent in this timestamp abstraction if the maximum of timestamps of its sources is less than or equal to the minimum of those of its targets and all of the timestamps of its targets are less than or equal to the validation timestamp of the relationship l.

5.4 Relationship between the standard interpretation and an abstraction

Suppose that s and t are states of the standard interpretation A and an abstract interpretation B, respectively. The changing of the states s in the standard interpretation reflects the developers' modifications to software documents and relationships. That is, the state s is in the "real content" level. In contrast, the changing of states t reflects the changes of abstract values for conformance tracking. In order for B to be a correct abstract interpretation of A, a relation must be defined between states s of the standard interpretation A and states t of an abstraction B such that the relation holds when t in B is viewed as a correct abstraction of s in A. That relation is called the abstraction relation $\Phi : A \rightarrow B$ in Gunter's build model. Our research adapts this notion to the hypertext model. The following soundness properties formally define the relation Φ for a hypertext:

Property 1 (Conformance soundness) *If* $\Phi(s,t)$ *and* $t \vdash_B l$ *then* $s \vdash_A l$. *That is, if* B *says that link* l *is consistent, then in the corresponding state* s *in* A*, the link* l *is also consistent.*

It is certain that this property is necessary since the purpose of the abstraction B is to provide an indication of a link's consistency. Before defining deletion soundness, we need to define a new notion. Given a function $f : X \rightarrow Y$ and a subset U of the domain of f. Let us denote the *restriction of* f to U by $f \mid U$.

Property 2 (Deletion soundness) *If* $\Phi(s,t)$ *then* $\Phi(s \mid U, t \mid U)$ *for any subset* $U \subseteq N$.

This property defines the soundness with respect to any deletion of nodes. That is, soundness needs to be maintained in any subset of a network. It is a natural requirement since any document could be removed from the system.

5.5 *"fix"* operation

The "fix" operation of the standard interpretation is used to correct an *inconsistent* link between nodes bridged by that link. After a fix, any node and link in the network can be modified. The set of those nodes is called *a change set*. In the build process, there are three types of changes with a rebuilding: a source file is modified; or target files are (re)produced; or a file is deleted. Therefore, a change set U in the build process either contains only source nodes or must satisfy $U = T_l$ for some causal link l. In contrast, in conformance analysis, both sources and targets can be produced or modified by human beings, so any nodes including link nodes can be in a change set. Formally, the "fix" operation takes as its arguments a link l, a state s in the standard interpretation A and returns a change set U and a new state s' in A. It is important to notice that the fix operation corrects one link at a time, together with connected nodes such that the following conditions are satisfied:

1. The link l is consistent in the new state s' (i.e., $s' \vdash_A l$),
2. If l is a causal link, $\neg\exists l' \in L$ such that $s \vdash_A l'$, $s' \nvdash_A l'$ and l' is connected to some node in S_l, and
3. If l is a non-causal link, $\neg\exists l' \in L$ such that $s \vdash_A l'$, $s' \nvdash_A l'$ and l' is connected to some node in A_l.

The first condition guarantees that after fixing, the link l is consistent in the new state. The second and the third conditions require that the fixing changes must not break the consistency of some neighboring links. This restriction is important to the soundness of the conformance analysis process that will be described later.

5.6 *"update"* operation

The purpose of the update operation is to record the new abstraction values of the current state in an abstract interpretation after the fix operation is invoked. The update function, β takes three arguments: 1) a change set U, 2) a state in standard interpretation s', and 3) an abstract state t. It returns a new abstract state t'.

The operational scenario for "fix" and "update" is as follows. Assume that s and t are states in the standard interpretation A and an abstraction B such that t is a correct abstraction of s (i.e., $\Phi(s,t)$ holds). Then, changes are made to some document nodes and s changes into s''. The update operation is then invoked so that t changes into t'' and new abstraction values are recorded. Suppose that document nodes are no longer in agreement with each other with respect to a link. When conformance analysis is requested, abstraction B says that non-conformance occurs in t''. The fix operation is invoked to correct the inconsistency of the link, in the standard intepretation, changing s'' into s'. Then, the update operation is now invoked so that t'' changes into t' and new abstraction values are recorded. The nodes that have been changed by the fix operation are in a change set U. After being updated, the new abstract state t' must be a correct abstraction of s'. Therefore, the function β must satisfy the following property:

Property 3 (Abstraction) *If* $s \mid (N \setminus U) = s' \mid (N \setminus U)$ *and* t *is a correct abstraction of* s *(i.e.,* $\Phi(s,t)$ *holds), then* $\Phi(s', \beta(U, s', t))$ *holds.*

This property is important since the new abstraction values need to be updated correctly in order for the abstraction B to maintain the ability to detect non-conformance. For example, in the timestamp abstraction, the function β, which is used to update timestamps, is defined as follows:

- For all nodes in the set $N \setminus U, \beta(U, s', t) = t$.
- For a node $x \in U, \beta(U, s', t) = 1+\max\{t_x \mid x \in H_x\}$ with $H_x = \{y \mid R(y, x)$ or $R(x, y)$ or $r(x, y)\}$.

The first rule says that there is no need to update abstract values (i.e. timestamps) for nodes that are not changed. The second rule records the timestamp for a changed node x such that the timestamp is greater than timestamps of any nodes related to x by any links.

5.7 Operational semantics

Operational rules represent the semantic operations of the hypertext model in which the states are changed and operations are invoked. They are defined based on the *marking* concept. Remember that a marking M on a hypertext network is a left-closed and target-closed subset of $N \cup L$. A marking M is said to be *consistent* with respect to a state s, if every causal link in M that has a target in M and every non-causal link in M that has all anchors in M are consistent. The set of documents is made consistent by applying a series of operations, link by link, that make additional documents consistent with the change and adds those updated documents to the marking. Operations need to be invoked in such a way that the consistency of the marking is always preserved. This conformance analysis process continues until no more documents can be added. First of all, a few notions need to be defined:

- Relative to a marking M, a causal link l has three possible states:
  - M contain all sources of l ($S_l \subseteq M$), but does not contain l or its targets. Let us denote this state by (M/l).
  - M contains the link l and its sources but none of its targets. That is, $l \in M$, $S_l \subseteq M$, and $T_l \cap M = \emptyset$. Let us denote this state by (l/M).
  - M contains all sources, targets and the link l. That is, $l \in M$, $S_l \subseteq M$, $T_l \subseteq M$. Let us denote this state by $(l \backslash M)$.
- Similarly, relative to a marking M, a non-causal link l has three possible states:
  - M contains at least one anchor of l but does not contain l. This state is also called (M/l).
  - M contains the link l and at least one (but not all) of its anchor nodes. That is, $l \in M$, $A_l \cap M \neq \emptyset$, and $A_l \not\subseteq M$. This state is also called (l/M).
  - M contains the link l and all its anchor nodes. That is, $l \in M$ and $A_l \subseteq M$. This state is also called $(l \backslash M)$.

Note that all other states are excluded by the left-closed and target-closed properties of a marking M.

Operational rules: Let us denote the evaluation relation by $\rightarrow$. An *evaluation state* is a triple consisting of a marking M, a standard state s and an abstract state t. It is denoted by (M, s, t). Let us denote the initial evaluation state by (M_0, s_0, t_0). M_0 consists of *source nodes*, i.e., the nodes that are not in the target set of any link. The initial s_0 contains assignments of the contents of document nodes to anchor nodes. Assume that the abstraction relation is Φ. Its "update" function β is invoked at this point on these initial source nodes to create an abstraction state t_0. The following rules formally define the conformance analysis process:

1. $\dfrac{(M/l) \quad t \not\vdash_B l}{(M, s, t) \rightarrow (M \cup \{l\}, s, t)}$ [Initiation-causal]
2. $\dfrac{(l/M) \quad (s', U) = fix(l, s)}{(M, s, t) \rightarrow (M \cup T_l, s', \beta(U, s', t))}$ [Execution-causal]
3. $\dfrac{(M/l) \quad t \vdash_B l}{(M, s, t) \rightarrow (M \cup \{l\} \cup T_l, s, t)}$ [Omission-causal]

(Note: U is the change set)

The initiation rule for causal links says that if the state t in the context of an abstraction B decides that l is *not* consistent, the marking M continues to spread out to include the link. The state of l relative to M now changes to (l/M), which sets up the application of the execution rule.

The execution rule for causal links says that when conformance analysis is requested and the marking M includes the link l, the "fix" operation is invoked to change s into s', then M spreads out to include the target set of l and the update operation β is invoked to update the abstraction values.

The omission rule for causal links says that if the state t in the context of an abstraction B decides that l is consistent, (i.e., the "fix" operation does not need to be invoked when conformance analysis is requested), M spreads out to include both the link and its target set.

Similarly, the operational rules for non-causal links are:

1. $\dfrac{(M/l) \quad t \not\vdash_B l}{(M, s, t) \rightarrow (M \cup \{l\}, s, t)}$ [Initiation-non-causal]
2. $\dfrac{(l/M) \quad (s', U) = fix(l, s)}{(M, s, t) \rightarrow (M \cup A_l, s', \beta(U, s', t))}$ [Execution-non-causal]
3. $\dfrac{(M/l) \quad t \vdash_B l}{(M, s, t) \rightarrow (M \cup \{l\} \cup A_l, s, t)}$ [Omission-non-causal]

5.8 The soundness theorem

To ensure the consistency among software documents, the process of conformance analysis must be sound. To

1. $\frac{(M/l) \quad t_i \not\vdash_{B_i} l \; for \; all \; i}{(M, s, t_1, t_2, ..., t_n) \rightarrow (M \cup \{l\}, s, t_1, t_2, ..., t_n)}$ [Generalized-Initiation-causal]

2. $\frac{(l/M) \quad (s', U) = fix(l, s)}{(M, s, t_1, t_2, ..., t_n) \rightarrow (M \cup T_l, s', \beta_1(U, s', t_1), \beta_2(U, s', t_2), ..., \beta_n(U, s', t_n))}$ [Generalized-Execution-causal]

3. $\frac{(M/l) \quad t_i \vdash_{B_i} l \; for \; some \; i}{(M, s, t_1, t_2, ..., t_n) \rightarrow (M \cup \{l\} \cup T_l, s, t'_1, t'_2, ..., t'_n)}$ [Generalized-Omission-causal]

where $t'_j = t_j$ if $t_j \vdash_{B_j} l$ and $t'_j = \beta_j(U, s', t_j)$ otherwise.

Figure 3. Generalized operational semantics

model the soundness of the process, a formal definition is needed. The definition is inspired by the soundness theorem in type systems described by Pierce [15]. Firstly, the concepts of *terminate* and *stuck* must be defined. The marking process is said to *terminate* if the initial marking M_0 continues to spread out and finally includes all nodes and links in the hypertext network. The process is said to become *stuck* if at a point different than terminate, there are no operational rules that can be applied.

The soundness is based on two properties: the process does not become stuck and the consistency of the marking is always preserved. Formally, the soundness theorem is expressed as two properties:

Property 4 (Progress) *A consistent marking never becomes stuck. Either it terminates or it can take a step according to the evaluation rules in the operational semantics.*

Property 5 (Preservation) *Let A be the standard interpretation and B be an abstraction. If M is a consistent marking, $(M, s, t) \rightarrow (M', s', t')$ with respect to a "fix" operation of A, and Φ is an abstraction relation from A to B with its update operation β, then M' is also a consistent marking in s', and t' is also a correct abstraction of s' (*i.e. $\Phi(s', t')$ *holds).*

These properties guarantees the conformance analysis process finally ends at a *terminate* and *consistent* marking including all nodes and links. Due to space limitation, the proof of this theorem is not described.

5.9 Combining multiple abstractions

Conformance analysis for documents can be automated depending on the built-in knowledge about the semantics of documents. In other words, different consistency management tools might provide different semantics dependent strategies for different types of documents and relationships that they support. A variety of technologies could be used to derive relationships and dependencies and to create conformance analysis strategies such as natural language processing (NLP), information retrieval, recognition, or knowledge engineering. The timestamp abstraction presented earlier is general, lightweight and can be used in combination with other techniques, while limitations on practical applications of NLP and recognition technologies prevent them from being used as the sole technique in a conformance analysis. The practical question is how techniques from different abstractions can be combined to yield an overall analysis process taking advantages of all abstractions together. A generalized operational semantics that allows for multiple abstractions is required.

Let us call A the standard interpretation. Let $\Phi_i : A \rightarrow B_i$ be the i^{th} abstraction relation of B_i ($i = 1...n$). Suppose that s is a standard state and t_i is the state of the abstract interpretation B_i such that $\Phi_i(s, t_i)$ holds. The evaluation state in operational rules is in the form of $(M, s, t_1, ..., t_n)$. Let the update operation of the abstraction B_i be β_i.

The generalized operational semantics are given in Figure 3. The first rule says that if all abstractions B_i decide that l is not consistent in their states t_i, the marking M continues to spread out to include the link, setting up the application of the execution rule. The execution rule says that when conformance analysis is requested and the marking M includes the link l, the "fix" operation is invoked to change s into s', then M spreads out to include the target set of l and all update operations β_i are also invoked on nodes in the change set to update the abstraction values for all abstractions. The third rule says that if *at least* one abstraction B_i decides that l is consistent in its state t_i (i.e., according to that abstraction B_i, the fix operation does not need to be invoked), M spreads out to include both the link and its target set. Except that abstraction, other abstractions update their values using their update operations. Similarly, the generalized operation rules for non-causal links and the generalized soundness theorem are developed.

6 Conformance analysis tools

6.1 Timestamp-based conformance analysis tool

We have implemented the conformance analysis model and integrated it into the Software Concordance environ-

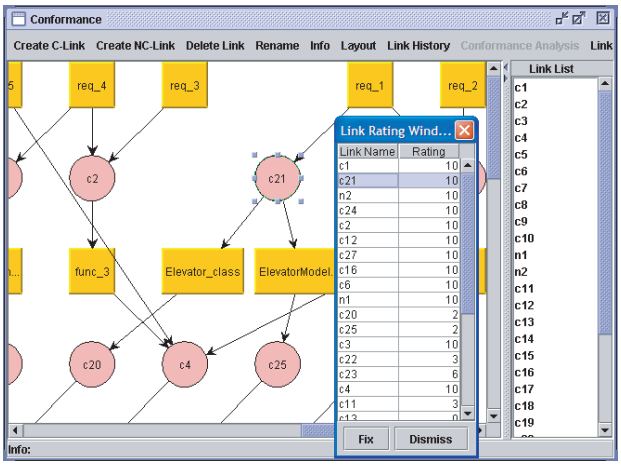

Figure 4. Timestamp strategy

ment (SC) [13]. The timestamp strategy described earlier has also been implemented. We also extended that strategy to include a heuristic *conformance rating* of the likely seriousness of conformance problems. Conformance ratings are made on a scale from zero (likely conformance) to ten (likely non-conformance). The conformance rating for a causal link is determined as follows. If $t_{src_{max}} \leq t_{dest_{min}}$ and $t_{dest_{max}} \leq t_{valid}$ then the destinations are newer than the sources and the link relationship was validated by a human sometime after the last changes to both. So, the rating is zero. In contrast, if $t_{src_{max}} \leq t_{valid} < t_{dest_{max}}$ then one of the destinations has changed since the last validation and the rating is five. We have arrived at these ratings only by human judgement. Empirical research is being conducted to provide a useful formula for these ratings.

Figure 4 shows the conformance analysis tool in action. The main window shows a conformance graph. The table lists links in an order in accordance to the conformance analysis process. The soundness theorem ensures that if users fix conformance problems in that order, a consistent marking will be achieved at the end of the process due to progress and preservation properties. Users can delay fixing of any link by just validating the link for a later examination.

6.2 Conformance analysis algorithm

To yield an overall conformance analysis process taking advantages of all abstractions together, we have implemented a conformance analysis algorithm that allows for the combination of multiple strategies. The algorithm is based on the generalized operational semantics in Figure 3. New conformance analysis strategies can be added into the system via a simple Java plug-in mechanism.

A strategy (i.e. an abstraction) must define the following functions: 1) *isApplicable(link)* (is the strategy applicable to a link and related documents?), 2) *isConsistent(link)* (according to the strategy, is a link consistent with connected documents?), 3) *evalRating(link)* (returning a rating for a link), 4) *updateOperation(U)* ("update" operation given a change set U), 5) *isFixOperation()* (returns the "fix" operation if the strategy knows how to fix a link and related documents, otherwise returns null). For example, for the timestamp strategy, *isApplicable* returns "true" for all links since it can be applied to any types of documents and its *updateOperation* updates timestamps of the link and document nodes that have been changed. However, the timestamp strategy does not provide *fixOperation*. For illustration, we have developed two other strategies. The *UML-validation* strategy evaluates the consistency of relationships between UML class diagrams and actual program source code that realizes them. Its *updateOperation* does nothing since it uses document contents as its abstract values. However, its *fixOperation* will regain the consistency between UML diagrams and source code. The other strategy is based on the vector-based model in information retrieval research. This strategy, which works only on textual documents, evaluates the conformance among non-program documents by their similarity measures.

The conformance analysis algorithm is shown in Figure 5. At any time, users can request to perform conformance analysis in a software project. The conformance graph is partitioned into separated subgraphs. The algorithm runs on each of those connected ones. Initially, the marking M contains only source nodes. The conformance analysis process continues until all nodes are visited (line 3). The set K contains all links that are at the state (M/l) (line 4). That is, K contains all causal links that are not in M and have all of their source anchors in M and all non-causal links that are not in M and have at least one of their anchors in M. If at least one strategy knows the consistency of the link, the link does not need to be fixed and the process will spread out to include the link and its anchor set. The rating for the link in this case is 0 (conformance). All abstractions that did not see the link consistent update their abstract values using their update operations (line 14). If all abstractions cannot tell the consistency of the link (line 15), an overall rating for the link l can be computed from the ratings from abstractions. In this case, "fix" operation needs to be invoked. If there exists one abstraction knowing how to fix the link and related documents, its "fix" operation will be called and the change set U is returned (line 17). Otherwise, the default "fix" operation in SC is invoked. The default operation presents the problem to users and asks them to fix it (line 18). Then, all abstractions update their abstract values (line 20). Finally, the marking M spreads out to include the link and its anchor set until the graph G is covered.

The extensibility of this conformance analysis tool and of the SC environment itself enables the use of different

```
(1) Assume that a graph G is connected
(2) M = all source anchor nodes
(3) While M has not covered the graph G (i.e. G ⊄ M)
(4)   K = {l | l ∉ M and ((l ∈ L_C, S_l ⊆ M) or (l ∈ L_N, A_l ∩ M ≠ ∅))}
(5)   For each link l in K:
(6)      Add link l to M
(7)      For all abstractions α that are applicable to link l:
(8)         Evaluate the rating of l by α.evalRating(l).
(9)      Is there any abstraction γ that recognizes the consistency of l?
(10)     Yes: Take the first one.
(11)          Assign rating of 0 to the link l.
(12)          For all abstractions a that did not see the consistency of l:
(13)            U = {link l}
(14)            a.updateOperation(U)
(15)     No: Compute the rating for link l from ratings from abstractions
(16)         Is there any abstraction χ that provides fixOperation?
(17)         Yes: Call U = χ.fixOperation(G)
(18)         No:Take the defaultFixOperation, which ask for users' help.
(19)            U = defaultFixOperation(G)
(20)         For all abstractions δ that are applicable to the link l
(21)            δ.updateOperation(U)
(22)     Spreading the marking M:
(23)     M = M ∪ T_l if l is causal or M = M ∪ A_l if l is non-causal.
(24)  EndFor
(25)EndWhile
```

Figure 5. Conformance Analysis Algorithm

consistency management frameworks. In addition, it also helps developers to avoid the *locality* problem in maintaining conformance among documents. In current practice, when discovering non-conformance, developers often focus on fixing a small part of a project and it might lead to the breaking of other consistency constraints. For example, a bug in a source file is found and then fixed. However, the new version of the source file might violate other consistency constraints established by a design or a requirement. Multiple bugs in several places can quickly make the matter complicated. The conformance analysis algorithm ensures the consistency for conformance graphs at the end of the conformance analysis process. Therefore, it lets developers focus on fixing local inconsistency while still ensuring the overall conformance afterward.

7 Conclusions

In this paper, we describe a formalism that models the conformance analysis among software artifacts in a software development process. The formalism can be used as a mathematical tool to determine the soundness of a conformance analysis strategy. With a sound strategy, the soundness theorem ensures consistency among documents at the end of a conformance analysis process. The generalization and extension of the formalism enable the combination of multiple consistency management strategies. The conformance model and its formalism provide the foundations for the implementation of conformance analysis tools and document relationship representations in the Software Concordance (SC) environment. The timestamp strategy with conformance ratings in SC is lightweight and can be used in combination with other strategies. The extensibility of the SC editors and conformance analysis mechanism provide infrastructures for our experiments of different consistency management methods. Future work includes experimental study on system usability and performance, and investigations on a more declarative approach for the definition of conformance analysis strategies.

References

[1] W. Chan, R. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, and J. Reese. Model checking large software specifications. *IEEE Transactions on Software Engineering*, 24(7):498–520, 1998.

[2] K. Chen and V. Rajich. RIPPLES: tool for change in legacy software. In *Proceedings of International Conference on Software Maintenance*, pages 230–239. IEEE, 2001.

[3] Document Object Model. http://www.w3.org/dom/.

[4] DrawSWF. drawswf.sourceforge.net.

[5] W. Emmerich, A. Finkelstein, C. Montangero, S. Antonelli, S. Armitage, and R. Stevens. Managing standards compliance. *IEEE Transactions on Software Engineering*, 25(6):836–851, 1999.

[6] S. Feldman. Make: A program for maintaining computer programs. *Software Practice*, 9:255–265, 1979.

[7] C. A. Gunter. Abstracting dependencies between software configuration items. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 9(1):94–131, 2000.

[8] A. Hunter and B. Nuseibeh. Managing inconsistent specifications: reasoning, analysis, and action. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 7(4):335–367, 1998.

[9] J. Koskinen. Empirical evaluation of hypertext access structures. *Journal of Software Maintenance and Evolution: Research and Practice*, 14(2):83–108, 2002.

[10] A. Kozlenkov and A. Zisman. Are their design specifications consistent with our requirements? In *Proceedings of Int. Conference on Requirement Engineering*. IEEE, 2002.

[11] C. Nentwich, W. Emmerich, and A. Finkelstein. Static consistency checking for distributed specifications. In *Proceedings of Automated Soft. Engineering*, pages 115–124, 2001.

[12] T. N. Nguyen and E. V. Munson. A model for conformance analysis of software documents. In *Proceedings of the Int. Workshop on Principles of Software Evolution*, 2003.

[13] T. N. Nguyen and E. V. Munson. The Software Concordance: A New Software Document Management Environment. In *Proceedings of ACM Conf. on Comp. Documentation*, 2003.

[14] B. Nuseibeh, S. Easterbrook, and A. Russo. Leveraging inconsistency in software development. *IEEE Computer*, 33(4):24–29, 2000.

[15] B. C. Pierce. *Type-Theoretic Foundations for Programming Languages*. MIT Press, 2000.

[16] I. Sommerville, P. Sawyer, and S. Viller. Managing process inconsistency using viewpoints. *IEEE Transactions on Software Engineering*, 25(6):784–799, 1999.

[17] G. Spanoudakis and A. Zisman. Inconsistency management in software engineering: Survey and open research issues. *Handbook of Software Engineering and Knowledge Engineering*, 1:24–29, 2001.

[18] Thorn UML editor. http://thorn.sphereuslabs.com/.

[19] W. F. Tichy. Smart recompilation. *ACM Transactions on Programming Languages and Systems*, 8(3):273–291, 1986.

[20] A. van Lamsweerde and E. Letier. Handling obstacles in goal-oriented requirements engineering. *IEEE Transactions on Software Engineering*, 26(10):978–1005, 2000.