Topological properties for characterizing well-formedness of Process Components

TRAN Dan Thu, TRAN Hanh Nhi, DONG Thi Bich Thuy
University of Natural sciences
227 Nguyen Van Cu, Q5, Hochiminh Ville, Vietnam
Email: tdt@hcmuns.edu.vn, thnhi@fit.hcmuns.edu.vn, thuy@hcmuns.edu.vn

Bernard COULETTE
UTM-IRIT
5, allées A. Machado F-31058 Toulouse, France
Email: Bernard.Coulette@univ-tlse2.fr

Xavier CREGUT
ENSEEIHT-IRIT
2, rue Camichel 31071 Toulouse, France
Email: Xavier.Cregut@enseeiht.fr

ABSTRACT
RHODES is an environment for modelling software processes, in which software processes are described by using a Process Modelling Language called PBOOL+. In this environment, a software process is built from PBOOL+ process components that can be reused to construct other processes.

To identify well-formed process components, we study properties to be able to characterize topological structure of the components. These properties should relate intrinsically to cohesion of a component, and coupling between components. We will consider two classical properties originated from graph theory, which are connection and transitive closure. These two properties are sometimes too strict to be applied, so we propose several weaker properties that are still useful for characterizing good components.

The paper aims to present these topological properties and their applications to reusable process components in context of the RHODES environment.

Keywords: process component, coupling, cohesion, process pattern, connection, closure

1. INTRODUCTION

In the last decade, Component-Based Software Engineering (CBSE) has become a very promising trend to improve software production process due to reuse of packaged components. Several industrial standards (COM, SOM, .NET, CORBA, and Java Beans) have been proposed for CBSE [Pree, 1997; Sametinger, 1997].

The concept of process component has recently been one of emerging subjects in the Software Process field, too. Process components and process patterns play an important role in a perspective of reuse software process. Some representative works in this domain can be mentioned such as Ambler's process patterns [Ambler, 1998], Catalysis patterns [D’souza et al. 1999], Process patterns for Componentware methodology [Bergner et al. 1998] and some metaprocesses [Chou& Chen, 2000; Coulette et al. 2002, Gary et al. 1998] proposed to define and reuse process components.

One of the most difficult challenges for both software components and process components is to evaluate their well-formedness. However, little contribution has been proposed so far for that purpose. Two primary properties that significantly influence the quality of components are low coupling in terms of few dependencies with other components, and high cohesion in terms of strong bindings between the elements within that component [Coad&Yourdon,1991; Sametinger,1997]. At the level of software components, most of works use the quantitative approach for software metrics to measure the coupling and cohesion degree of software products [Fenton, 1991; Badri, 2003]. With regards to process components, the literature is poor. We have not found any significant metrics or other propositions for evaluation process components well-formedness.

Motivated by the lack of research in assessing quality of process component, we have studied the problem of characterizing reusable process components. We use the qualitative approach based on topological properties to characterize well-formed process components.
This generic approach is a necessary step to derive news metrics for process components from proposed properties. Our work is illustrated with the process components in the RHODES project, a PSEE developed at the IRIT\(^1\) laboratory [Coulette et al. 2000].

In this paper, we focus on topological consistency and its application to identify well formed process components. Section 2 recalls concepts of RHODES process components. Then we describe topological properties in section 3 and their application to identify well-formed process components in section 4. Practical experiment of our approach is presented in section 5 with a case study. Finally, section 6 discusses related works, and the last section gives some conclusions and perspectives.

2. SOFTWARE PROCESS COMPONENTS IN RHODES

We define informally a process component as “a coherent and reusable unit of processes”. Each process component is an entity that can be used when constructing a software process.

In order to describe the concept of process component, in the RHODES environment we developed a Process Modelling Language called PBOOL\(^+\) [Coulette et al. 2000] and a component metamod [Coulette et al. 2001]. We also proposed a metaprocess [Coulette et al. 2002] to model and to reuse the process components.

2.1 Metamodel of process components

The two levels of process components considered in RHODES metamod are: elementary process components and complex process components. Figure 1 presents our metamod for process components.

Elementary process components correspond to atomic entities of a process: activity, product, role and strategy [Crégut&Coulette, 1997]. Elementary components can share a common specification. This property aims to support the component evolution: each component can be replaced by another one that has the same specification.

Every type of software product is modelled by a product model. Every product possesses subproducts and operations that are similar to attributes and methods of classes in an object-oriented programming language. The Role model describes the role of developers who participate in the development. A development process is generally decomposed into activities that fit well together to implement this process. An activity model is an elementary component that models a development activity. Every activity describes the work specification through input and output products, and assertions (pre-condition, post-condition, and invariant) and the role of the developers who can perform the activity.

---

1 Informatic Research Institute of Toulouse
The activity implementation describes realization sketches (each of them is a way to realize the activity) and methodological guidance (to correct inconsistencies). A strategy helps developers choose among ways to perform an activity. Complex process components [Coulette et al. 2001] are used to modularize elementary ones for the aim of process reuse and evolution. A complex component has one specification and one or several implementations. Its specification contains a set of elementary component specifications. Each complex implementation contains one elementary implementation for each elementary specification of its complex specification.

2.2 PBOOL+: Language for describing process components

PBOOL+ [Coulette et al. 2000] is a Process Modelling Language that was defined in the RHODES environment to formally describe component-based software processes. Its predecessor, the PBOOL language (Process-Based Object Oriented Language [Crégut & Coulette, 1997]), was inspired from the Eiffel language with certain extended functionalities to be able to model enactable software processes. This language supports mainly the modelling of elementary process components. We have improved PBOOL to take into account complex components that support modularity and genericity of described processes.

The PBOOL+ language – an improved version of PBOOL, has been proposed to model reusable and evolvable process components. This section introduces characteristics of PBOOL+, and some concrete examples of process component modelling.

2.2.1 Characteristics of PBOOL+

The PBOOL+ language maintains the following important characteristics of PBOOL:

- support of possibility to describe different ways (called “sketches”) to perform each enacting activity, an important property for dynamic evolution of processes.

Besides, the PBOOL+ language proposes certain factors to go forwards low coupling between process components and high cohesion of each process component. Specifically, it supports following characteristics:

- Separation between specification and implementation: this characteristic allows to separate specification of a component from its detailed implementation;

- Modularization of process components for the purpose of reuse and evolution of these components: elementary components which are strongly related to each other can be grouped into a complex component;

- Support of parameterized activities: each activity can take a generic parameter that can be instantiated by an effective activity; this property allows to describe generic process components, such as process patterns. A process pattern (Ambler 1998; Bergner et al. 1998; Coulette et al. 2000) is a reusable generic process whose goal is to provide a solution to recurring situations occurring in the domain of software process modelling. We have adapted some properties of existing design patterns (Gamma 1995) and process patterns (Ambler 1998) to formalize the description of process patterns. Thus, a RHODES process pattern (Coulette et al. 2001) is described by a name, an intention, a context, a semi-formal solution, a formalisation, and an application clause.

2.2.2 Example of process components

We choose some process components extracted from a simplified software change process (Figure 2) for illustration. The whole process will be considered further in following sections of the paper.

First, process components are conceptually modelled in UML-like formalism². This graphical view presents relationships between elementary process components. Then, to make components executable, the UML-like description is formalized by using PBOOL+.

\[\text{In order to represent both dynamic and static aspects of process in one activity diagram, we propose using a new notation named “weak-fork” to describe the decomposition of an activity. The sub-activities of a decomposed activity by “weak fork” are not strictly parallel in their execution.}\]
The main products manipulated by the Software Change Process are documents. Figure 3 illustrates a UML class diagram of these products. Designers can apply design patterns [Gamma et al. 1995] in modelling. In this example, both two patterns “Bridge” and “Composite” are used in the class diagram for the products. The former is used to decouple business characteristic (left side of the diagram) of manipulated products from their structural characteristic (right side of the diagram). On the left side, relationships between several kinds of software documents are
presented: each document type is derived from a generic document. On the right side, the “Composite” pattern – due to its recursive property – is used to describe the hierarchical structure of document content.

Similarly, activities participating in the process are also described in UML-like formalism. Figure 4 presents decomposition of an activity into subactivities. The “MAKE_CHANGES” activity is decomposed into three activities: “MODIFY_CODE”, “TEST_UNIT”, and “INTEGRATE_CHANGES”. Subactivities are parallel in their execution with synchronization constraints (dependencies) that are represented by dotted arrows. In fact, this order is established due to input/output products, preconditions and postconditions of each activity, which are shown more clearly in the PBOOL+ description of activities.

The formalized description in PBOOL+ of the “MAKE_CHANGES” activity is illustrated in Figure 5. It can be seen in this figure the separation between specification and implementation of an activity: the activity "MAKE_CHANGES” has a unique specification named “MAKE_CHANGES_SP”. This activity requires the role “DEVELOPER”, it means that a person who plays the role of a developer in the software change process may perform this activity.

The precondition “READY” requires that the change plan is ready before executing the activity; the postcondition “FINISH_CHANGE” ensures that source code is successfully finished after executing the activity. In the implementation part of this activity, there are two exclusive realisation sketches: “CALLTOOLS” and “DECOMPOSITION”. At enactment time, the developer chooses one of them to perform the activity. Each sketch describes a (heuristic) way to perform the activity.

Elementary process components can be grouped together in order to define a reusable complex component. For example, the complex component “MAKE_CHANGES_COMP” (Figure 6) contains a set of elementary specifications (INTEGRATE_CHANGES_SP, MAKE_CHANGES_SP, MODIFY_CODE_SP, TEST_UNIT_SP…), and an implementation “MAKE_CHANGES_COMP_IMP” containing the elementary components (INTEGRATE_CHANGES, MAKE_CHANGES, MODIFY_CODE, TEST_UNIT, …) which correspond to the above elementary specifications.
2.3 RHODES – An environment for modelling process components

Environment RHODES has been developed to support modelling and enacting software processes based on process components. RHODES comprises mainly three groups of constituents supporting the RHODES metaprocess: structural and consistency constraints, process description and management constituents, and process enactment constituents. Figure 7 illustrates the functional architecture of RHODES.

Figure 5. Description in PBOOL+ of an activity

Figure 6. Grouping some elementary components into a complex component

Figure 7. Architecture of RHODES

The aim of structural and consistency constraints is to control process components described by process designers through the metaprocess. The important constituent is the RHODES metamodel (Section 2.1) that defines the structure of process components and relationships between them. Moreover, a set of properties is proposed to control consistency of process components, and to characterize their well-formedness.

The description and management of software processes are supported by two main constituents: the Process Component Editor (PCE) and the Process Component Base (PCB). The latter is implemented over an Object Database of Jasmine System [Khoshafian et al. 1999]. The RHODES metamodel is used to establish classes of the object database. The above consistency constraints are implemented as methods of those classes. The PCE establishes the interaction between process designers and PCB during the progress of process component modelling. Functionalities of PCE are driven by the RHODES metaprocess and controlled by the structural and consistency constraints.

Process enactment in RHODES is realized by the execution kernel, the PBOOL+ compiler, and executable instances of RHODES. The kernel, which is implemented by Eiffel classes, aims to mechanize fundamental activities of software processes. These activities are independent from any application domain, and are controlled strictly by predefined predicates. Each software process
described in PBOOL+ is compiled by the PBOOL+ compiler to be linked with the execution kernel, so that this process becomes an executable instance. The executable instance is an enactable process for a specific application domain, which can be enacted by software developers to produce different products of software life cycle. Consistency of elaborated products is controlled by the predefined predicates and specific predicates defined in the PBOOL+ process.

3. TOPOLOGICAL PROPERTIES OF PROCESS COMPONENTS

To ensure the consistency of products elaborated by a software process, the software process itself must be a consistency one. Besides, process inconsistency should be detected as soon as possible. Verifying a process before its execution can eliminate many inconsistencies that might happen at the execution time.

As stated in the introduction, in this paper we focus on topological consistency that includes several consistency properties concerning topological relationships between process components. We first present some concepts and definitions related to topological structure of process components.

3.1 Topological structure of complex components

The above UML description (Figure 3) of the software documents presents naturally a topological structure. We can simplify graphical notations and maintain the topological structure: Figure 8 shows a directed graph (the graph G1) corresponding to the UML diagram. The elementary components and their relationships are represented respectively by graph vertices and arcs. Each arc of the graph is assigned a relationship type, called a “stereotype”. For example, the vertices “SP”, “DE” and “TE” correspond to three kinds of documents SPECIFICATION, DESIGN and TEST PLAN; the arcs joining these three vertices with the “DO” vertex correspond to specialized documents of the generic document; these arcs possess the stereotype “specialisation”.

Generally, for each complex component CC, G(CC) denotes the graph corresponding to its UML diagram. This graph is also called the associated graph of CC. In specific situations, there exist also needs to investigate the graph with only arcs of the same stereotype. For example, to detect cycles of the inheritance relation, we consider the graph with arcs that have a stereotype “inheritance”: it is denoted by G(CC)/{“inheritance”}.

![Figure 8. Associated graph of process components](image)

Suppose $c_1, c_2 \in CC$, we define the followings relations:

i. If there exists an arc that joins $c_1$ and $c_2$ we say that $c_1$ depends directly on $c_2$ and denote this relation by $c_1 \rightarrow c_2$.

ii. When the dependence of $c_1$ on $c_2$ is established indirectly by the elements of CC: $c_1 \rightarrow x_0, x_0 \rightarrow x_1, x_1 \rightarrow x_2, \ldots, x_k \rightarrow c_2$; in which $x_1, x_2, \ldots, x_k \in CC$, it means that there is a directed path in CC from $c_1$ to $c_2$; we say that $c_1$ depends indirectly on $c_2$. If $c_1$ depends directly or indirectly on $c_2$, we denote this relation by $c_1 \sim c_2$.

iii. If there exists the elements $x_0, x_1, x_2, \ldots, x_k \in CC$ so that $x_1 \rightarrow x_0$ or $x_i \rightarrow x_{i+1}$ and $c_1 \sim x_0, x_k \sim c_2$ for all $i = 1, 2, \ldots, k$, we say that there is a chain connecting $c_1$ and $c_2$. If $c_1 \sim c_2$ or there exist a chain between $c_1$ and $c_2$, we denote $c_1 \sim c_2$. This relation is reflexive, symmetric and transitive.

We can deduce that $c_1 \sim c_2$ implies $c_1 \sim c_2$ and for any z in CC, $z \sim c_1$ and $z \sim c_2$ implies $c_1 \sim c_2$.

3.2 Topological consistency properties

In contrast to the metric approach that tries to quantify the cohesion and coupling of components, we study topological properties that can characterize the coupling between components and the cohesion of a component, especially process components.

As design characteristics, coupling is the degree of interdependency between components, and cohesion is the degree of connectivity between its individual elementary components [Steven et al. 1974]. The objective of good design
is to reach components having low coupling and high cohesion [Eder et al, 1992, Sametinger, 1997].

Two families of topological properties investigated are connection properties and closure ones. Transitive closure property has been largely used in many domains, for instance in database [Zloof, 1976; Aho et al.1979; Cruz et al. 1987; Yannakakis, 1990; Houtsma et al. 1991; Schwarz et al. 1998], configuration management [Christensen, 2000] and network [Khuller&Raghavachari, 1996; Olivera et al. 2003]. In Component-based Software Engineering, some models also suggest using transitive closure as an attribute that guarantees the low interdependency of component [OMG, 2002; Sametinger3, 1997]. Thus we propose to apply closure property to characterize the component coupling. Regarding the cohesion, with the definition of Steven, the relation between the cohesion concept and the connection of the elements in a component can be seen naturally. Moreover, inspired by some works in software metrics that base on the connection property to develop their measures for cohesion [Li&Henry, 1993; Hitz&Montazeri, 1995, Chae et al., 2000], we consider connection as a main property that can characterize the component cohesion.

3.2.1 Connection properties

Naturally, designers build a complex component by grouping elementary components in a way such that they are related somehow to each other. It is not reasonable to put components not having any connection into a group. An important issue is how to characterize the connection property between components that are grouped together. We first recall classical connection types that were defined in graph theory [Berge, 1967], and then propose weaker types of connection that can be used more appropriately to study the collaboration degree between elementary components in a complex component.

3.2.1.1 Classical connection properties

The very strict connection for directed graphs is strong connection (also called “connection” in very few documents investigating only directed graphs). This type of connection is defined as follows: “a graph is strongly connected iff it exists a directed path between every pair of vertices”. When applying to a complex component, this property requires that for any two elementary components \( c_1, c_2 \) of the complex component, \( c_1 \) and \( c_2 \) are related tightly together: \( c_1 \sim c_2 \) and vice versa. This is the highest degree of component cohesion.

For example, the associated graph \( G(CC1) \) of the component \( CC1 \) (Figure 9), which is generated from a set of three activities \{Realize_Chg, Validate_Chg, RefuseSW\} extracted from the simplified software changed process (Figure 2), is strongly connected. Therefore, it can be concluded that \( CC1 \) is highly coherent\(^4\). Particularly, a directed cycle of elementary components is strongly connected and therefore highly coherent too\(^5\).

![Figure 9. Component strongly connected and its associated graph](image)

However, strong connection is a too strict property: it is difficult to require an associated graph of a complex process component to have this property. The component \( CC1 \) in Figure 9 is the unique component (containing more than one activity of the process) of the whole process (Figure 2) that has the strong connection property.

\(^3\) Sametinger uses the terminology "self-contained" with the same meaning of "transitive closure"

\(^4\) For the reason of legibility, in the rest of this article process components will be described by UML-like diagrams without their associated graphs

\(^5\) Here we don’t discuss about the reasonableness of those cyclic components, we just argue that if the designer has to (or love to) make such a solution because of some particular semantics or context of his problem, that component should include all elementary components to obtain high cohesion (cf. definition of strong connection).
Nevertheless, it is not reasonable in practice to design a component such as CC1, because the activity Validate_Chg requires both the sub-activities AcceptSW and RefuseSW for its adequacy (Figure 2). Generally, instead of CC1, designers would create the component CC2 with \{Realize_Chg, Validate_Chg, AcceptSW, RefuseSW\} (Figure 10) for reuse of the activity Validate_Chg. CC2 is not strongly connected (because a directed path from AcceptSW to RefuseSW doesn't exist) but we can feel intuitively that it has somewhat a good cohesion. There are many similar cases to this example in reality, so it is necessary to find out weaker types of connection that can be applied more suitably for process components.

A property, which is slightly weaker than strong connection, is strong semi-connection property (or unilateral connection). A graph G is strongly semi-connected (or unilaterally connected) iff for any two vertices of the graph, at least one is reachable due to a directed path from the other.

![Diagram](image)

**Figure 10. Component unilaterally connected**

For example, the above component CC2 is not strongly connected but it is unilaterally connected. The details of this are shown in Table 1.

<table>
<thead>
<tr>
<th>(c_1, c_2)</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>Realize_Chg, Validate_Chg</td>
<td>Realize_Chg\rightarrow Validate_Chg</td>
</tr>
<tr>
<td>Realize_Chg, AcceptSW</td>
<td>Realize_Chg\rightarrow Validate_Chg\rightarrow AcceptSW</td>
</tr>
<tr>
<td>Realize_Chg, RefuseSW</td>
<td>Realize_Chg\rightarrow Validate_Chg\rightarrow RefuseSW</td>
</tr>
<tr>
<td>Validate_Chg, AcceptSW</td>
<td>Validate_Chg\rightarrow AcceptSW</td>
</tr>
<tr>
<td>Validate_Chg, RefuseSW</td>
<td>Validate_Chg\rightarrow RefuseSW</td>
</tr>
<tr>
<td>RefuseSW, AcceptSW</td>
<td>RefuseSW\rightarrow Realize_Chg\rightarrow Validate_Chg\rightarrow AcceptSW</td>
</tr>
</tbody>
</table>

*Table 1. Detail of paths in CC2*

The main difference between unilateral and strong connection is that there exists at least one directed path connecting two vertices instead of two directed paths (one from \(c_1\) to \(c_2\) and the other from \(c_2\) to \(c_1\)). In the case of CC2, it has a path from RefuseSW to AcceptSW but does not have a reverse path.

The connection property that is usually applied in many domains is weak connection (that is simply called “connection” in various papers). This type of connection requires that any two different vertices of the graph must be linked by a sequence of linked successive arcs without considering direction of arcs. For example, the component MAKE_CHANGES_SW (Figure 11) with four activities \{Make_Chg, Modify_Code, Test_Unit, Integrate_Chg\} is neither strongly connected nor unilaterally connected because there is no directed path between “Modify_Code” and “Test_Unit”.

![Diagram](image)

**Figure 11. Component weakly connected**

However, if we don't consider the direction of arcs, there is a chain connecting these two elements: Modify_Code \(\leftarrow\) Make_Chg \(\rightarrow\) Test_Unit, hence, MAKE_CHANGES_SW is (weakly) connected. The relationship between two elements in a connected component is less tight than that in a strongly or unilaterally connected component.

Another connection property is quasi-strong connection: for any two vertices of a graph, there exists a third vertex from which the two formers are reachable by directed paths. In fact, by a classical result for finite graphs [Berge, 1970, p.30], this connection property is equivalent to property of rootedness: there exists a common vertex from which the others are reachable by directed paths.

For example, the whole software change process (Figure 2) is rooted at SW_Change; the “MAKE_CHANGES_SW” component (Figure 11) is rooted at Make_Chg; the component CC2 shown above is rooted at three vertices: Realize_Chg, Validate_Chg, and RefuseSW.
The above classical connection properties can be formally defined as follows:

**Definition 1.** Let $CC$ be a complex component

**Strong connection.**

$$\forall x, y \in CC \ (x \sim y \land y \sim x)$$

**Unilateral connection.**

$$\forall x, y \in CC \ (x \sim y \lor y \sim x)$$

**Quasi-strong connection.**

$$\forall x, y \in CC \ (\exists z \in CC \ (z \sim x \land z \sim y))$$

**Rootedness.**

$$\exists r \in CC \ (\forall x \in CC \ (r \sim x))$$

**Connection.**

$$\forall x, y \in CC \ (x \sim y)$$

Then, we can deduce from these definitions the following proposition that also comes from classical results in graph theory [Berge, 1970].

**Proposition 1.** Let $CC$ be a complex component.

(i) $CC$ is quasi-strongly connected if and only if $CC$ is rooted.

(ii) We have the following sequence of implication about connection properties of $CC$:

- strong connection $\Rightarrow$ unilateral connection
- unilateral connection $\Rightarrow$ rootedness
- rootedness $\Rightarrow$ connection

**Proof.**

(i) **Rootedness $\Rightarrow$ Quasi-strong connection**

Suppose $CC$ has a root $r$.

For all $x, y \in CC$, we choose $z=r$ and have $z \sim x$ and $z \sim y$. So $CC$ is quasi-strongly connected.

(ii) **Strong connection $\Rightarrow$ Unilateral connection**

Suppose $CC$ is strongly connected. For all $x, y \in CC$, we have:

$$(x \sim y \land y \sim x) \Rightarrow (x \sim y \lor y \sim x)$$

So $CC$ is unilateral connected.

(iii) **Unilateral connection $\Rightarrow$ Rootedness**

Suppose $CC$ is unilaterally connected. For all $x, y \in CC$, we have $(x \sim y \lor y \sim x)$

If $x \sim y$ then put $z=x$ else put $z=y$

So we have $z \sim x$ and $z \sim y$ for all $x, y \in CC$, and therefore $CC$ is rooted.

**Rootedness $\Rightarrow$ Connection**

Suppose $CC$ has a root $r$.

For all $x, y \in CC$, we have $r \sim x$ and $r \sim y$ which implies $x \sim y$.

So $x \sim y$ for all $x, y \in CC$, and therefore $CC$ is connected.

3.2.1.2 Weaker types of connection properties

In practice, sometimes, the connection between elements in a system is not explicitly represented. For example, the component $CC_3$ below (Figure 13) generated by $\{Design_{Chg},$
Modify_Code, Integrate_Chg\} is disconnected: the subgraph \(G(CC3)\) contains no arcs, it is totally discrete. However, intuitively, it can be seen that there are certain business relationships between those elements of CC3. In fact, although \(G(CC3)\) is not connected but it is a subgraph of a connected graph: the whole software change process is connected (moreover it is rooted) and contains CC3.

Therefore we propose a property called “under-connection”, a trivial one in mathematics but useful in practice. A graph is under connected if it is a subgraph of a connected one. Thus the component CC3 is under connected. We suggest that each complex process component must have at least the under connection property, because it is unreasonable to build a component from elementary components that are not related to each other at all.

![Figure 13. Component under connected](image)

By investigating the structure of complex components, we recognize a natural property of connection that is weaker than connection but stronger than under connection. This property, called “connection by familial proximity”, is more suitable in many practical situations of designing complex components.

Firstly, the familial relation, denoted by ‘\(\approx\)’, is defined as: for any two vertices \(x, y \in CC\), \(x \approx y\) if and only if:

\[x = y \text{ or } (x \text{ depends on } y) \text{ or } (y \text{ depends on } x) \text{ or } (a \text{ third vertex depends on both } x \text{ and } y)\]

(both \(x\) and \(y\) depend on a third vertex).

In this definition, the third vertex does not necessarily belong to CC. It is easy to see that the relation \(\approx\) is reflexive and symmetric (i.e. for all \(x, y\) we have \(x = y\) and \(x \approx y \Rightarrow y = x\)). When \(x \approx y\), \(x\) and \(y\) are called “in family”.

Then we define the familial proximity relation denoted by ‘\(\cong\)’ as follows: for any two different vertices \(x, y \in CC\), \(x \cong y\) iff it exists a chain of vertices \(z_1, z_2, ..., z_k\) in CC so that:

\[z_i = z_j, y = z_k, z_i \approx z_{i+1} \text{ for all } i = 1, 2, ..., k-1.\]

(We emphasize that the condition “all \(z_k\) are in CC” is important in this definition.) The relation \(\cong\) is reflexive, symmetric and transitive.

A complex component is connected by familial proximity if and only if \(x \cong y\) for any two vertices \(x, y \in CC\); CC is connected by family if and only if \(x \cong y\) (\(x\) and \(y\) are in family) for any two vertices \(x, y \in CC\). For example, the above component CC3 is under connected but it is not connected by familial proximity: although the two elements Modify_Code and Integrate_Chg are in family (because of Make_Chg→Modify_Code and Make_Chg→Integrate_Chg), however, Design_Chg and Modify_Code are not related by the ‘\(\cong\)’ relation in context of CC3. In fact, we have:

\[
\begin{align*}
\text{Design_Chg} & \approx \text{Realize_Chg}, \\
\text{Realize_Chg} & \approx \text{Make_Chg}, \\
\text{Make_Chg} & \approx \text{Modify_Code},
\end{align*}
\]

but Realize_Chg and Make_Chg do not belong to CC3.

Let’s consider now the component CC4 generated by \(CC3 \cup \{\text{Realize_Chg, Make_Chg}\}\) (Figure 14). CC4 is not connected (because Design_Chg is separated from the others in CC4). Nevertheless, CC4 is connected by familial proximity: thanks to the following relationships between its five elements:

\[
\begin{align*}
\text{Design_Chg} & \approx \text{Realize_Chg}, \\
\text{Realize_Chg} & \approx \text{Make_Chg}, \\
\text{Make_Chg} & \approx \text{Modify_Code}, \\
\text{Modify_Code} & \approx \text{Integrate_Chg},
\end{align*}
\]

we can deduce that \(x \cong y\) for any two elements \(x, y \in CC4\). CC4 is not connected by family, but the component with three elements \(\{\text{Design_Chg, Planning_Chg, Realize_Chg}\}\) is.
These connection properties can be formally defined for a complex component CC as follows:

**Definition 2.** Let CC be a complex component.

- **Under connection.**
  \[(\exists CC_1 \text{ connected})(CC \subseteq CC_1)\].

- **Connection by familial proximity.**
  \[ (\forall x, y \in CC) \ (x \equiv y) \].

- **Connection by family.**
  \[ (\forall x, y \in CC) \ (x \approx y) \].

The following proposition can be deduced from the above definitions.

**Proposition 2.** Let CC be a complex component.

(i) We have following sequence of implications about connection properties of CC:

- Connection \(\Rightarrow\) Connection by familial proximity \(\Rightarrow\) Under connection

(ii) The connection by family implies the connection by familial proximity.

**Proof.**

- **Connection \(\Rightarrow\) Connection by familial proximity**
  We have \(x \equiv y \iff [x = y \lor x \rightarrow y \lor y \rightarrow x \lor (\exists z, z \rightarrow x \land z \rightarrow y) \lor (\exists z, x \rightarrow z \land y \rightarrow z)]\)
  So \([x = y \lor x \rightarrow y \lor y \rightarrow x] \Rightarrow x \equiv y \ (1)\)

- **Connection by familial proximity \(\Rightarrow\) Under connection**
  For all u, v if \(u \equiv v\), we have:
  \([u = v \lor u \rightarrow v \lor v \rightarrow u \lor (\exists z, z \rightarrow u \land z \rightarrow v) \lor (\exists z, u \rightarrow z \land v \rightarrow z)]\)
  which implies \(u = v\) or there exists a chain from \(u\) to \(v\).

Now suppose CC is connected, for all \(x, y \in CC\), \(x \sim y\).
There exists \(x_0, x_1, \ldots, x_k \in CC\) so that:
\(x_0 = x, x_k = y, \text{ and } [x_i = x_{i+1} \lor x_i \rightarrow x_{i+1} \lor x_{i+1} \rightarrow x_i, \forall i = 1, 2, \ldots, k]\)
Then by (1) we have:
\(x_0 = x, x_k = y, \text{ and } [x_i = x_{i+1}, \forall i = 1, 2, \ldots, k] \Rightarrow x \equiv y \ (by \ definition \ of \ \equiv)\)
So for all \(x, y \in CC\), \(x \equiv y\), i.e. CC is connected by familial proximity

**Connection by familial proximity \(\Rightarrow\) Under connection**
For all \(u, v\) if \(u \equiv v\), we have:
\([u = v \lor u \rightarrow v \lor v \rightarrow u \lor (\exists z, z \rightarrow u \land z \rightarrow v) \lor (\exists z, u \rightarrow z \land v \rightarrow z)]\)
which implies \(u = v\) or there exists a chain from \(u\) to \(v\).
For all \(x, y\), if \(x \equiv y\) then there exists \(z_0, z_1, \ldots, z_k\) so that: \(x = z_0 \approx z_1 \approx \ldots \approx y = z_k\)
So \(x = y\) or there exists a chain from \(x\) to \(y\).
Now, suppose CC={\(c_1, c_2, \ldots, c_m\)} is connected by familial proximity (\(c_i \neq c_j\) if \(i \neq j\)). For each \(k = 1, 2, \ldots, m-1\), there exists a chain from \(c_k\) to \(c_{k+1}\).
Let CC₁ be the of all elements of these chains.
We have CC⊆CC₁ and CC₁ is connected, so CC is under-connected.

(ii) **Connection by familial \(\Rightarrow\) Connection by familial proximity**
Because \(x \equiv y\) implies \(x \equiv y\), we deduce immediately this property.

### 3.2.2 Closure properties

To achieve low coupling between components, each component should be autonomous, it should not much depend on other components. Intuitively, to be independent, a component must be “sufficient” somehow.

The sufficiency can be characterized by topological closure properties. A classical property concerning topological closure is “transitive closure”. A transitively closed component must contain all elements on which an element of the component depends. A component with this property is completely independent; it does not depend on any component and can be reused independently from other components.

---

6 The elements of these chains may belong to CC or not
For example, the “MAKE\_CHANGES\_SW” component above (Figure 11) is transitively closed; however the component CC2 (Figure 10) is not, because Accept SW \( \in \) CC2 and Accept SW depends on Deliver SW, but Deliver SW \( \notin \) CC2; moreover Realize Chg depends on Make Chg and its subactivities (Figure 2), which not belong to CC2. To be transitively closed, CC2 must take Deliver SW and all four elements of component “MAKE\_CHANGES\_SW”, that means CC5 = CC2 \( \cup \)”MAKE\_CHANGES\_SW”\( \cup \) {Deliver SW} is transitively closed.

However, it is not convenient to satisfy the transitive closure property in some cases, because of taking all necessary elements, a transitively closed component can contain too many elements, and can make that component becomes low coherent. A component must not “cut” any directed cycle, i.e. if the component shares at least an element with a cycle then it must comprise the whole cycle. A component CC is closed relative to cycles if for each directed cycle of arcs, the component CC must not depend on any outside element to which a certain element of the component depends.

For each component CC, Dependence(CC) is defined as the set of elements not belonging to CC such that for one of those there exists an element depending on this one; in case CC contains only one element, for instance CC = \{c\}, we will write Dependence(c) instead of Dependence(\{c\}). Obviously, CC is transitively closed if and only if Dependence(CC) = \( \emptyset \). If Dependence(CC) \( \neq \emptyset \), every element of CC either depends on no element, or depends on another element of CC, or depends on an element of the set Dependence(CC). We also say that CC is closed relatively to Dependence(CC). For example, in the case of CC2, we have

\[
\text{Dependence} (CC2) = \{\text{Make Changes}, \text{Deliver SW}\},
\]

that negates the transitive closure of CC2 and CC2 is closed relatively to \{Make Chg, Deliver SW\}.

We study other properties weaker than transitive closure but still useful for characterizing the relative independence of each process component. These properties are local closure, closure relative to cycles, and closure relative to paths that are being introduced in following paragraphs.

The local closure property aims to characterize the local sufficiency at each element of a complex component. Let’s consider the component CC6 (Figure 15) generated by

\[
\{\text{Agree}_\text{Req}, \text{Design}_\text{Chg}, \text{Realize}_\text{Chg}\},
\]

that is not transitively closed. CC6 does not comprise any element on which Design Chg or Realize Chg depend. However, Agree Req belongs to CC6 and depends on three elements: Design Chg, Planning Chg, Realize Chg; two of those belong to CC6 but Planning Chg does not. Therefore, CC6 is “more sufficient” if it comprises Planning Chg. In fact, we are going to see that the component CC7 = CC6 \( \cup \) {Planning Chg} satisfies condition of a local closure (Figure 16).

![Figure 15. Component not transitively closed](image)

![Figure 16. Component locally closed](image)

A component CC is locally closed if, for each c contained in CC, Dependence(c) is either included in CC, or has no common elements with CC. In the case of CC7, for example, we can determine Dependence(c) for each c \( \in \) CC7, and see that:

\[
\text{Dependence(Agree}_\text{Req}) \subset CC7, \text{Dependence(c)} \cap CC7 = \emptyset, \forall c \neq \text{Agree}_\text{Req}
\]

Therefore, it can be concluded that CC7 is locally closed.

Closure relative to cycles requires that the component must not “cut” any directed cycle, i.e. if the component shares at least an element with a cycle then it must comprise the whole cycle. A component CC is closed relatively to cycles if for each directed cycle of arcs, the component CC either contains all or no vertices of the cycle. For example, the above component CC7 is locally closed but it is not closed relatively to cycles because CC7 contains Realize Chg that is an element of the following cycle:

\[
\text{Realize}_\text{Chg} \rightarrow \text{Validate}_\text{Chg} \rightarrow \text{RefuseSW}
\]

but CC7 does not contain the whole cycle.
A component $CC$ is *closed relatively to paths* if for each directed path of arcs, if the component $CC$ contains source and destination of the path then it must contain all vertices of the path. Because each element of a cycle is not only source but also destination, if a component is not closed relative to cycles then it will not be closed relative to paths. For instance, $CC7$ is not closed relatively to paths.

The following definition presents formal description of the four closure properties.

**Definition 3.** Let $CC$ be a complex component.

(i) $Dependence(CC) = \{e \in CC \mid \exists e \in CC; c \rightarrow e\};$

(ii) $Transitive closure$

$Dependence(CC) = \emptyset$

(iii) $Local closure$

$(\forall e \in CC)$

$(Dependence(c) \subset CC \lor Dependence(c) \cap CC = \emptyset)$

(iv) $Closure relative to cycles$

$(\forall$ directed cycle $\mu) (\mu \subset CC \land \mu \cap CC = \emptyset)$

(v) $Closure relative to paths$

$(\forall$ directed path $\mu$ from $x$ to $y)$

$(x \in CC \land y \in CC \Rightarrow \mu \subset CC)$

From this definition, we can prove the following proposition that describes the logical relation between the closure properties.

**Proposition 3.** Let $CC$ be a complex component.

(i) The transitive closure property implies all other closure properties.

(ii) The closure relative to paths implies the closure relative to cycles.

**Proof.**

(i) *Suppose $CC$ is transitive closed, we have:*

$Dependence(CC) = \emptyset$ \hspace{1cm} (I)

Prove that $CC$ is locally closed:

If $(\exists e \in CC)(Dependence(c) \subset CC$

$\land Dependence(c) \cap CC = \emptyset)$

we will show a contradiction.

Because of $Dependence(c) \subset CC$, we can choose $e \in Dependence(c)$ and $e \notin CC$,

So $c \rightarrow e$, $c \in CC$, $e \in CC$

$\Rightarrow e \notin Dependence(CC)$ which contradicts (I)

Hence $(\forall e \in CC)(Dependence(c) \subset CC$

$\lor Dependence(c) \cap CC = \emptyset)$

which says that $CC$ is locally closed.

Prove that $CC$ is closed relatively to paths:

Suppose $CC$ is not closed relatively to paths, we will show a contradiction.

There exists a directed path $\mu$ from $x$ to $y$ so that $x$, $y \in CC$ and $\mu \notin CC$.

So there are $u$, $v$ so that $u \in CC$, $v \notin CC$ and $u \rightarrow v$

The last statement implies $v \in Dependence(CC)$ which contradicts (I).

Prove that $CC$ is closed relatively to cycles:

We can deduce immediately this statement by the above and the statement (ii).

(ii) *Closure relative to paths $\Rightarrow$ Closure relative to cycles*

Suppose $CC$ is not closed relatively to cycles, we will prove that it is not closed to paths.

There exists a directed cycle $\mu$ so that $\mu \subset CC$ and $\mu \cap CC = \emptyset$.

So we can choose $b \in \mu \cap CC \Rightarrow b \in \mu \land b \in CC$.

Because $\mu$ is a cycle we can write it as follows:

$\mu = b \rightarrow x_{1} \rightarrow x_{2} \rightarrow \ldots \rightarrow x_{k-1} \rightarrow b$

Therefore $\mu$ is also a directed path from $b$ to $b$, in which $b \in CC$ but $\mu \notin CC$.

And so $CC$ is not closed to paths.

**3.2.3 Properties relative to stereotypes**

The simplified software change process in Figure 2 is considered only for the relation “decomposition” of activities. That is, this presentation of the process corresponds to a graph of type $G(CC)/(\langle decomposition\rangle)$. In general, let $ST$ a non-empty set of arc stereotypes, the graph $G(CC)/ST$ is the subgraph taking all vertices of $G(CC)$, whose each arc must have a stereotype $st \in ST$. For example, Figure 17 shows the limitation of the graph $G1$ (Figure 7) when considering stereotypes $ST = \{\langle aggregation\rangle, \langle composition\rangle\}$. $G1$ is connected but $G1/ST$ is not connected: we say that $G1$ is not connected when limiting to the set $ST$ of stereotypes.

Let’s consider the subgraph $G2$ generated by $\{Document (DO), Doc. structure (DS), Chapter (CH)\}$. $G2$ is still connected (moreover it is unilaterally connected) when limiting to the set $ST$. 

14
Now, we can define the topological properties relative to a set of stereotypes as follows.

**Definition 4.** Let $CC$ be a complex component, $ST$ a stereotype set, and $\rho$ one of connection or closure properties. We say that $CC$ satisfies the property $\rho$ relative to $ST$ iff the graph $G(CC)/ST$ satisfies the property $\rho$.

Especially, the component $CC$ is unilaterally connected relatively to stereotype «decomposition» if the graph $G(CC)/\{\text{«decomposition»}\}$ is unilaterally connected. $CC$ is transitively closed relatively to stereotype «composition» if $G(CC)/\{\text{«composition»}\}$ is transitively closed.

For instance, in Figure 18 the component $CC8 = \{\text{Section, Simple section, Complex section}\}$ is not transitively closed in general because it misses the element “Paragraph”. However, when limiting to stereotype «specialisation», $CC8$ is transitively closed.

**Proposition 4.** Let $CC$ be a complex component, $ST1$ and $ST2$ two stereotype sets, $\rho1$ one of connection properties and $\rho2$ one of closure properties.

(i) if $ST1 \subset ST2$ and $CC$ satisfies the property $\rho1$ relative to $ST1$, then $CC$ also satisfies the property $\rho1$ relative to $ST2$.

(ii) if $ST1 \subset ST2$ and $CC$ satisfies the property $\rho2$ relative to $ST2$, then $CC$ also satisfies the property $\rho2$ relative to $ST1$.

**Proof.**

Suppose $ST1$, $ST2$ are two stereotype sets and $ST1 \subset ST2$, $CC$ is a complex component. We only demonstrate some representative cases. The other cases can be developed similarly.

(i) **Connection relative to $ST1$ ⇒ Connection relative to $ST2$**

Suppose $CC$ is connected relatively to $ST1$. For all $x, y \in CC$, there exists a chain in $CC$ from $x$ to $y$ so that each arc of the chain has stereotype $s \in ST1$.

Because of $ST1 \subset ST2$, all stereotypes of arcs of the chain also belong to $ST2$. So $CC$ is connected relatively to $ST2$.

(ii) **Transitive closure relative to $ST2$ ⇒ Transitive closure relative to $ST1$**

Suppose $CC$ is transitively closed relatively to $ST2$. We have:

$\text{Dependence}_{ST2}(CC) = \{e \notin CC / \exists c \in CC: c \rightarrow e \text{ and stereotype of the arc } c \rightarrow e \text{ is in } ST2\} = \emptyset$

Because of $ST1 \subset ST2$, we also have:

$\text{Dependence}_{ST1}(CC) = \{e \notin CC / \exists c \in CC: c \rightarrow e \text{ and stereotype of the arc } c \rightarrow e \text{ is in } ST1\} = \emptyset$

So $CC$ is transitively closed relative to $ST1$.

In the case that $ST2$ is the set of all possible stereotypes, it can be seen that $G(CC)/ST2$ and $G(CC)$ are the same. So, the following corollary is deduced directly from the proposition.

**Corollary 4.** Let $CC$ be a complex component, $ST$ a stereotype set, $\rho1$ one of connection properties and $\rho2$ one of closure properties.

(i) if $CC$ satisfies the property $\rho1$ relative to $ST$, then $CC$ also satisfies the property $\rho1$.

(ii) if $CC$ satisfies the property $\rho2$, then $CC$ also satisfies the property $\rho2$ relative to $ST$.

The corollary says that each connection relative to stereotypes always implies the respective
connection in general. Besides, each closure property implies the respective closure relative to any set of stereotypes.

Specifically, we can deduce some results from the above proposition and corollary.

(a) If \( CC \) is connected relatively to a stereotype \( st \), then \( CC \) is connected relatively to any stereotype set \( ST \ni st \).

(b) If \( CC \) is connected relatively to a stereotype \( st \), then \( CC \) is connected.

(c) If \( CC \) is connected by familial proximity relatively to a stereotype \( st \), then \( CC \) is connected by familial proximity relatively to any stereotype set \( ST \ni st \).

(d) If \( CC \) is connected by familial proximity relatively to a stereotype \( st \), then \( CC \) is connected by familial proximity.

(e) If \( CC \) is transitively closed, then \( CC \) is transitively closed relatively to any stereotype set.

(f) If \( CC \) is locally closed, then \( CC \) is locally closed relatively to any stereotype set.

(g) If \( CC \) is closed relative to cycles (or paths), then \( CC \) is also closed relative to cycles (or paths) when limiting to any stereotype set.

4. APPLICATION OF TOPOLOGICAL PROPERTIES

This section presents an application of the above topological properties in context of process modelling. Our aim is to guide the process designers to obtain the well-formed components that satisfy the principle “low coupling between complex process components, high cohesion of each complex process component” for the purpose of reusing or storing these components.

To illustrate for this application, we consider some components extracted from the Software Change Process (Figure 19), note that here we describe deliberately the relationship of the whole process and its products - the documents - via the stereotype “manipulate”.

Suppose that a process designer makes the following components :

\[
CC_a = \{ \text{Analyse}_Chg, \text{Paragraph}, \text{Chapter} \}, \\
CC_b = \{ \text{Submit}_Chg, \text{Make}_Chg, \text{DeliverSW} \}, \\
CC_c = \{ \text{Design}_Chg, \text{Realize}_Chg, \text{Validate}_SW \}, \\
CC_d = \{ \text{Design}_Chg, \text{Planning}_Chg, \text{Realize}_Chg \}.
\]

We suggest applying the above connection properties and closure properties to characterize the component cohesion and coupling.

The cohesion of a complex component concerns the degree of cooperation between its individual elementary components. Every complex component should have a cohesion degree as high as possible. Based on propositions in Section 3.2.1, we can use the following connection properties respectively to assess different levels of cohesion from highest to lowest: strong connection, unilateral connection, rootedness, connection, connection by family, connection by familial proximity, under connection.

Intuitively, we can see that the structuralization of \( CC_a \) is not coherent: \( \text{Analyse}_Chg \) and \( \text{Paragraph} \) have no relation (when limiting to the stereotype "decomposition"). \( CC_b \) is more reasonable because its elements belong to the same process, however, it is “too far”\(^7\) to reach an element of \( CC_b \) from each other. \( CC_c \) is better than \( CC_b \): its elements, activities are collaborative together to change and evaluate the software. In four components above, \( CC_d \) is the most adherent: its activities are systematically related.

In fact, by verifying connection degree of these components, we have the same results as follows:

- \( CC_a \) is not under connected,
- \( CC_b \) is under connected,
- \( CC_c \) is connected by familial proximity,
- \( CC_d \) is connected by family

Suppose that \( \text{Cohesion}(CC) \) is the cohesion degree of a component, thus, we can conclude :

\[
\text{Cohesion}(CC_a) < \text{Cohesion}(CC_b) < \text{Cohesion}(CC_c) < \text{Cohesion}(CC_d)
\]

The component coupling concerns the degree of interdependency between components. Process designers should try to obtain a component coupling degree as low as possible. The closure properties can be applied to characterize the coupling.

\(^7\) For the limited subject of this paper, we don’t expand deeply into the concept of distance between the vertices of a graph. We informally use this concept of distance as the maximum number of arcs to traverse from a vertex to another in a component. Thus, in the figure 19 we can observe: from any vertex of \( CC_d \) we can reach another vertex of \( CC_d \) by traversing 2 arcs; the maximum number of arcs that we have to traverse in \( CC_c \) is 3 (from \( \text{Design}_Chg \) to \( \text{Validate}_SW \)); but in \( CC_b \), to reach \( \text{DeliverSW} \) from \( \text{Submit}_Chg \) we must traverse 7 arcs.
Figure 19. A simplified software change process with product.
A component having the transitive closure property is completely independent from other components: it is self-sufficient. For instance, the whole simplified software change process is transursively closed. The component \{Make_Chg, Modify_Code, Test_Unit, Integrate_Chg\} is smaller but still transursively closed. In the above example, considering only the relation “decomposition”, we observe that:

- CC_c is neither transursively closed nor locally closed
- CC_d is not transursively closed but locally closed

Thus, it can be stated that component CC_d is more sufficient than CC_c.

We have proposed guidelines to apply the connection properties and closure properties to each kind of RHODES complex components. Table 2 and Table 3 present respectively extractions of the synthesis of these guidelines on connection properties and closure properties.

The four degrees of guidance proposed are necessary, recommendatory, optional, and impossible:

- the necessary degree and the impossible one come from the semantics of PBOOL+ components. For example, a set of products PBOOL+ cannot be strongly connected (“impossible”) because every product is defined based on specifications of products rather than on the other products. On the contrary, this set must be under-connected (“necessary”) because it is a subset of a certain process that must be connected;

- the recommendatory degree is proposed from our experience when evaluating process components of the RHODES environment. A component that doesn’t satisfy a necessary property maybe still valid (according to the syntax rules of PBOOL+) but it isn’t well-formed (according to the requisition of that property). For instance, a process component should be connected by familial proximity (“recommendatory”) because it is not reasonable to group elementary components which are not related nearly;

- the optional degree allows the process designer to decide according to his heuristics. Therefore, an optional property can be applicable in a certain context but not in the others. For example, in order to obtain a generic and high reusable component, CC_d is designed without comprising the detail elements of Design_Chg and Realized_Chg. Thus, CC_d is just locally closed, not transitively closed. In the case that we need to have CC_d as an autonomous component that can be reused independently, we should have to include the subactivities of Design_Chg and Realize_Chg in CC_d to make it become transitively closed.

Process designers can use the guidelines in the Table 2 on assessing the cohesion of a process entity.

<table>
<thead>
<tr>
<th>Categories</th>
<th>Properties</th>
<th>Process patterns</th>
<th>Process components</th>
<th>Processes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strong connection</td>
<td>Optional</td>
<td>Optional</td>
<td>Impossible</td>
<td></td>
</tr>
<tr>
<td>Unilateral connection</td>
<td>Optional</td>
<td>Optional</td>
<td>Optional</td>
<td></td>
</tr>
<tr>
<td>Rootedness</td>
<td>Optional</td>
<td>Optional</td>
<td>Necessary</td>
<td></td>
</tr>
<tr>
<td>Connection</td>
<td>Necessary</td>
<td>Optional</td>
<td>Necessary</td>
<td></td>
</tr>
<tr>
<td>Connection by familial proximity</td>
<td>Necessary</td>
<td>Recommendatory</td>
<td>Necessary</td>
<td></td>
</tr>
<tr>
<td>Under connection</td>
<td>Necessary</td>
<td>Necessary</td>
<td>Necessary</td>
<td></td>
</tr>
<tr>
<td>Connection by family</td>
<td>Optional</td>
<td>- Optional - Recommendatory for an activity set</td>
<td>Impossible</td>
<td></td>
</tr>
</tbody>
</table>

Table 2. Application guidelines of connection properties

Similarly, process designers can use the guidelines in Table 3 to assess the independence degree of process entities. To decrease the component coupling, a component should satisfy a certain closure property. For example, for each process component, it is recommended to have the property of closure relative to cycles.

It should be noted that each property can imply another property. In both tables (Table 2 and Table 3), we present the degrees of guidance either in bold typeface or in normal typeface. A normal-formatted degree means that it is deduced from a stronger property that is possessed by the component.

For instance, because it is “necessary” for the rootedness property of the process, so the connection, connection by familial proximity property and the under connection property are “necessary” too.
5. CASE STUDY

In order to test out our approach in a real context, we applied the proposed topological properties to the process “Model with VUML”\(^8\), which was used to elaborate a multidimensional model of the system in the project “Distant Learning System” at University of Toulouse 2, France.

In this section, we will report how the designer applied the guidelines in Table 2 and Table 3 to evaluate process components extracted from the process “Model with VUML” for the purpose of reusing and storing them.

In this process, there are many activities having an interactive loop (or a constrained loop). These activities have the same manner of repeat, they just differ in repeated activities. Therefore we defined and applied two parameterized process patterns (Figure 20) for modelling those activities. The “Loop interactively” process pattern requires a repeated activity as unique parameter, while the “Loop constrainedly” process pattern has two parameters – an activity to repeat and a product list that help determine when to stop that activity. Figure 21 and figure 22 describe respectively the process “Model with VUML” in UML-like notation and its corresponding graph.

Among the components created by the designer, we choose the most meaningful components to illustrate the application of proposed properties. For each component, we introduce firstly the original design, then its evaluation by using the topological properties, and finally the designer’s modifications on that component.

\(^{8}\) VUML is an object-oriented design and analysis method integrating UML and the point of view concept [Nassar et al., 2003]
Figure 21. “Model with VUML” process
Figure 22. The associated graph of process “Statically model with VUML”
However, in regard to the closure property, CC10 doesn’t satisfy the requirement because it is neither transitively closed nor locally closed (cf. Table 3).

To assure the local closure for this component, the designer added into CC10 two activities Identify & Structuralize UCs and Realize UCs and obtained CC11 (Figure 25). But this would change the original intention of this component.
Finally, he decided to separate the component C11 into two components to obtain the following well-designed reusable components:
- A rooted and transitively closed C12 for preliminary investigating the system (Figure 26).
- A rooted and locally closed C13 for constructing the use cases (Figure 27).

Moreover, he could have a third reusable component CC14 for modelling use cases by combining CC12, CC13 and the activity Model_Usecases (Figure 28). CC14 is also a well-formed process component because it reaches “Rootedness” and “Local closure”.

The second component that we examine is CC15. The designer intended to create this component for grouping the specific activities of viewpoint modelling approach into an independent component. Firstly, he only chose the following activities (Figure 29)

\[ CC15 = \{\text{Identify Viewpoints, Creat ViewpointsModel, Elaborate DiagramVUML}\} \]
**Evaluation:** The proposed component is only under connected and locally closed, hence we cannot reuse it as an independent component.

To make this component better for the intended purpose, its cohesion and closure degree should be raised respectively to “Connection by familial proximity” and “Transitive Closure”.

The designer attempted making a connected (by familial proximity) component CC16 by adding into CC15 the following activities: Identify_ActorsViewpoints, Construct_UCs, Model_UCs, and Elaborate_MultiviewsModel (Figure 30). However, CC16 just reaches the “Closure relative to path”. To make CC16 become transitively closed, the component must include all the dependencies of its activities. We have:

- Dependence(Identify_ActorsViewpoints) = \{Identify_ActorTypes, Structuralize_Actors, Identify_Viewpoints\};
- Dependence(Construct_UCs) = \{Identify_ActorsViewpoints, List_Requirements, Identify&Structuralize_UCs, Realize_UCs\};
- Dependence(Model_UCs) = \{Preliminarily_Investigate, Construct_UCs\};
- Dependence(Elaborate_MultiviewsModel) = \{Resolvle_Inconsistencies, Elaborate_VUMLDiagram\};

Excluding the activities existing in CC16 from the set of dependencies, these following activities were added to CC16:

- Preliminarily_Investigate, Identify_ActorTypes, Structuralize_Actors, List_Requirements, Identify&Structuralize_UCs, Realize_UCs, Loop_Constrainedly, Create_ActorViewpointsModel, Resolvle_Inconsistencies

In their turns, the dependencies of new activities were added in CC16 too. For instance, CC16 must further include Dependence(Preliminarily_Investigate) = 

- \{Analyse_CurrentSystem, Model_BusinessProcess, Create_Glosarry\} and so on.

Consequently, we get a new component CC17 containing almost all activities of the process except the activity Model_VUML. By adding this root activity to CC17, the designer raised the cohesion of CC17 to “Rootedness” (Figure 31 and cf. Figure 21).

Due to this evaluation, the designer decided to replace CC15 by CC17.

---

9 Regarding to the “\textit{instantiate}” relation, the process pattern “\textit{Loop constrainedly}” and the parameter “\textit{Create viewpoints model for an actor}” are in the dependence of Create_ViewpointsModel.
The component CC18 was firstly composed for the purpose of drawing the diagrams. The reuse of such a component is necessary in all UML-based processes. Its elements are (Figure 32):

\[
\text{CC18} = \{ \text{Draw}_{-}\text{Actortypes}, \text{Draw}_{-}\text{UCs}, \\
\text{Add}_{-}\text{UseRelations}, \text{Add}_{-}\text{SpecRelations}, \\
\text{Add}_{-}\text{ExRelations}, \text{Draw}_{-}\text{Classes}, \\
\text{Draw}_{-}\text{Relations} \}
\]

**Evaluation**: Component CC18 is under connected and transitively closed, hence it satisfies the minimum condition of cohesion, and attains the highest requirement for closure.

To make CC18 obtain “Connection by familial proximity” the following activities must be added to CC18: Realize_UCs, Construct_UCs, Model_UCs, Elaborate_MultiviewsModel, Elaborate_DiagramVUML and Draw_Diagram (Figure 33). However, when doing so, CC18 is no longer transitively closed.

But if we add more activities to CC18 to reach low-coupling, it will make the component become too large and not specific for the original intention.

Therefore, instead of that solution, the designer chose to separate the activities of CC18 into two following components with more specific objective:
- CC19 = \{ \text{Draw}_{-}\text{Actortypes}, \text{Draw}_{-}\text{UCs}, \\
\text{Add}_{-}\text{UseRelations}, \text{Add}_{-}\text{SpecRelations}, \\
\text{Add}_{-}\text{ExRelations} \} for drawing the Use cases diagram (Figure 34)
- CC20 = \{ \text{Draw}_{-}\text{Classes}, \text{Draw}_{-}\text{Relations} \} for drawing Class diagram (Figure 35)

This time, both CC19 and CC20 are connected by family and transitively closed. These well-formed components can be appropriately reused in many processes.
Figure 32. An under connected and transitively closed process component

Figure 33. CC18 is modified to reach “Connection by familial proximity”
We also evaluated the process pattern “Loop interactively” to verify whether it is well-designed for using as an autonomous component.

**Evaluation**: This pattern has very high-cohesion and low-coupling because it is unilaterally connected and transitively closed (Figure 34).

The last component considered is the whole Process “Model with VUML” as CC17 (cf. Figure 31).

This evaluation was made because the designer wanted to know whether it is necessary to put the process patterns together with other activities of this process.

As shown in Figure 22, the associated graph of process is not connected when limiting to stereotype “decomposition” because the process patterns vertices are unreachable from other activities vertices in the graph. However, when the process patterns are excluded, the graph becomes rooted. In fact, if component CC17 includes those patterns, its cohesion is lower than without them.

Because the examined process patterns don’t have any specific semantic, they are often stored as basic components to be reused by various processes. Thus, we can ignore such generic patterns when evaluate a component in regard to closure property. Therefore, we can say that CC17 is transitively closed even when it excludes those patterns.

Finally, the designer decided that it had better to put the process patterns outside the process. This decision is reasonable and often taken in reality.

The chosen examples in this section are just some of components that we have evaluated by using the topological properties. Preliminary results show that the proposed approach is useful, but this has to be confirmed by further empirical studies.

### 6. RELATED WORKS

Some representative projects have introduced process components such as Pynode [Avrilionis et al. 1996] and OPC (“Open Process Components”, Gary et al. 1998). However, these environments do not propose topological properties to characterize component cohesion and coupling.

The SPEM metamodel submitted to OMG (Object Management Group, 2002) also introduces the process component concept. Except the transitive closure property, SPEM does not propose other topological properties for process components.

Process pattern-based approaches (Ambler, 1998; Bergner et al., 1998; D’souza and al., 1999) pay attention to the collection of reusable process
components due to experience rather than studying structuralization of those.

At the software product level, several formal models are proposed for componentware technology. The Bergner’s model [Bergner et al., 2000] defines formally concepts concerning componentware such as: interface, connection, type and instance. The authors also model the dynamic changing of a component system over discrete time. Perry [1996] was concerned about the well-formedness of system compositions and component specification. The well-formedness is based on constraint predicates (preconditions, postconditions or obligations) associated to component interfaces.

Zaremski and Wing [1997] studied the matching between component specifications. The definitions of matching specification also capture the notions of generalization/specialization and substitutability of software components. However, these contributions do not pay attention to properties that can characterize topologically well-structuralized components.

There are also several metrics built to evaluate software components in regards to the coupling property as NCR [Lorenz&Kidd, 1994], CBO, RFC [Chidamber&Kemerer, 1994], CF [Abreu et al., 1993], and cohesion property as LCOM [Li&Henry, 1993; Chidamber&Kemerer, 1994], CR [Balasubramanian, 1996], CAMC [Bansiya et al., 1999].

Among those metrics, LCOM3 (Lack of Cohesion in Methods) is redefined in terms of graph as a number of connected components of a graph [Hitz&Montazeri, 1995]. This makes it a generic metric that coincides with the original definition [Chidamber&Kemerer, 1994] if the graph nodes represent methods, and the graph edges represent attributes accessed. There are other attempts also based somewhat on graph properties to derive new metrics: the cohesion measure of Chae et al. [2000] takes into account the members that have actually impact on the cohesiveness of a class and is defined in terms of the degree of the connectivity among those members; Badri [2003] also used a graph to represent the methods in a class, then defined a new metric in terms of the relative number of related methods, however, focused on methods invocation criterion, and introduced the concept of indirect usage of attributes. Tom Mens [2002] proposed a metamodel based on type graph representation of object-oriented software to define a set of generic metrics. But this work did not consider cohesion and coupling metrics.

7. CONCLUSION

In this paper, we suggest applying topological properties for assessing well-formedness of components. We have studied the two families of classical properties connection and closure. Because those classical properties are sometimes too strict to be used for identifying well-formed process components, we have proposed some new properties related to connection and closure and presented its application for assessing process components.

We have provided process designers with some guidelines to evaluate the coupling between process components and the cohesion in each process component. These topological properties have been developed in the context of the RHODES project whose goal was to integrate reusable process components.

Our work contributes a new approach to the process component evaluation that is still poorly discussed in literature. The main advantage of our approach is that it provides a generic way to assess the process components at the design level. The definitions and results concerning the proposed topological properties are developed in a generic context. They do not depend on semantic aspects of a specific environment. Therefore, although being proposed to be used for process components, these topological properties can also be adapted for the context of software components. Moreover, we hope that new metrics may be derived from these properties.

We are now applying the results of this work in building the RHODES process components repository. That is, we follow the proposed guidelines based on topological properties to obtain well-designed process components. Another future work is implementing the topological properties in term of new metrics embedded in a process component editor – a tool to be integrated in the RHODES environment to support methodological guidance to process designers.

REFERENCES


Mens, T. Lanza, M., 2002, "A Graph-Based Metamodel for Object-Oriented Software Metrics". Electronic Notes in Theoretical Computer Science, Volume 72, Number 2, 2002


Stevens, W., Myers, G., and Constantine, L., 1974, Structured Design, IBM Systems Journal, 13(2) 115-139, 1974

