A Taxonomy of Orthogonal Properties of Software Architectures

Lars Brattahl, Per Runeson
Department of Communication Systems, Lund University, Sweden.
(lars.brattahl | per.runeson)@itn.hb.se

Abstract. There are several suggestions regarding the definition of the term software architecture. Yet discussions concerning the subject are ambiguous. This leads to unclear discussions and slow scientific progress. This paper suggests a coherent set of properties - a Taxonomy of Orthogonal Properties of Software Architecture (TOPSA) - that enables more precise communication within the software architecture domain. The TOPSA has been empirically validated by using it during an industrial development project. The paper shows some uses of the TOPSA in education and system development. It is concluded that the TOPSA is applicable for a wide range of uses. The TOPSA sharpened the conceptual tools available in the domain, resulting in more precise presentation, retrieval and use of knowledge within the software architecture domain.

1. Introduction

Several authors have defined software architecture, although differently [4, 5, 29]. Models that illustrate types/ views of software architecture have been suggested by [3, 22, 30], yet the terms and the views are used differently by different authors. This results in studies being hard to compare, and confusion in communication illustrates the need for further investigation into the subject. For the practitioner, it is a challenge to know which views and models should be made when.

An example of the confusion is apparent when, for example [4], states that a software architecture is completely visible in code, while [15] states that a software architecture is an interpretation, not entirely visible in code. Depending on what view of software architecture is being discussed, both may be correct.

Another example comes from the performance computation domain; object oriented techniques usually considered "good" cannot be used in the code due to performance constraints. Creating large amounts of objects during execution may be even less permissible. However, during the analysis phase, the software architecture may use object-oriented constructs, that are not retained in actual code. In this paper, two types of software architecture are discussed: Conceptual software architectures during domain analysis, and realization of software architecture during design.

A third example is the use of the term "abstraction level". Does this refer to the aggregation level of components, or to the conceptual level of the architecture? This paper attempts to relate both of these concepts into a coherent set.

The ambiguity is a problem, since it through linguistic determinism [14] affects the human ability to describe entities related to software architecture, as well as it restricts how humans can reason about them. In order to facilitate discussions within the software architecture domain, a set of properties of software architecture is suggested. The set of properties has the following characteristics:

1. Orthogonal definitions. The definitions given within the set are orthogonal. Thus the properties defined can be disjointly combined.
2. Process independent. It is possible to use the set for discussing both forward and reverse-engineering processes.
3. Representation independent. The set neither assumes nor dictates any particular representation.

In section 2 a Taxonomy of Orthogonal Properties of Software Architecture (TOPSA) is introduced. In sections 3 and 4, a number of uses of the TOPSA for research, education and development purposes are demonstrated. Finally, some related work are discussed, and some further work is outlined.

2. A Taxonomy of Orthogonal Software Architecture Properties

2.1 Introduction

There are several definitions of software architecture, for example [4, 5, 29]. The basis for the TOPSA is the definition of software architecture given in [4]: "The software architecture of a program or computing system is the structure or structures of the system, which comprise software component, the externally visible properties of those components, and the relationships among them."

Copyright 1999 Lars Brattahl, Per Runeson
This definition is used in this paper due to its widespread availability via [4]. In this paper, a number of properties are added to this definition. Below they are stated in an informal form. In section 2.2, the definitions are more formally defined.

**Abstraction level.** A conceptual software architecture holds a representation of a model of the architecture, with the purpose of explaining or understanding how functional requirements are allocated to collaborating structures. A realisational software architecture is directly visible in a representation of a model that corresponds directly to source code. An example of a conceptual architecture can be a single cellular telephone communicating with a single base station, while a realisational architecture would show the physical entities involved in the communication. A product can have several conceptual architectures, each of which can be mapped to adjacent abstraction levels.

**Dynamic.** A static software architecture is present in the code. A dynamic software architecture is the result of executing the code. The difference between dynamic architecture and static architecture is determined by what changes its state. The static architecture is changed by evolution and development activities, while the static architecture, time, and external stimuli govern the state of a dynamic architecture. An example of a static architecture are the structures and relations present in code that create dynamic software processes, while the structures and relations created when executing the code constitute the dynamic architecture.

**Aggregation level.** Aggregation level refers to if and to what extent a structure is made from other structures. A low aggregation level implies that the structure is composed using a short aggregation tree, and a high aggregation level implies that a structure is composed from one or several structures which in turn are made from one or several structures - a deep aggregation tree. These concepts are illustrated in figure 1.

Together, these properties form the TOPSA space, which is illustrated in figure 2.

![Fig. 2. TOPSA space](image)

Thus the TOPSA space has three dimensions: Abstraction level, dynamism and aggregation level.

### 2.2 Definitions

**Abstraction level: Conceptual/Realisational.** A realisational software architecture for a software system is directly visible in the code.
A conceptual software architecture may be visible in the code, but it is not necessarily so. During forward engineering a conceptual architecture holds a representation of a model of the software architecture before implementing it in source code, while during reverse engineering, the conceptual architecture is an interpretation of the code, that is not necessarily directly visible in the code.

There is at all times a transformation function between a conceptual architecture and the realizable architecture (1).

\[ C(n) \rightarrow R(n) \] (1)

\[ R(n) \rightarrow C(n) \] (2)

On the other hand, (2) does not necessarily hold true as the conceptual architecture is an interpretation if recreated during reverse engineering. However, there always exists at least one \( Z_s \).

As there can be several levels of conceptual architecture, (3) holds true, while (4) does not necessarily hold true.

\[ C_s(0) \rightarrow C_s(1) \rightarrow \ldots \rightarrow C_s(N_s) \] (3)

\[ C_s(0) \rightarrow C_s(1) \rightarrow \ldots \rightarrow C_s(N_s) \] (4)

**Dynamism:** Static/dynamic. A dynamic software architecture is defined in (5).

\[ \begin{align*}
A(0) \\
1 > t_0
\end{align*} \] (5)

\( t_0 \) equals a point in time where an instruction pointer points at the first instruction in the software, but the first instruction has not yet been executed. For example, the instruction pointer can reside in a computer or in the mind of someone interpreting the software in this manner.

The dynamic software architecture is illustrated in Figure 3.

![Dynamic Software Architecture Diagram](image)

Fig. 3. Static and dynamic architectures. E(t) is external stimuli. The time \( t \) is an example of external stimulus. User interaction is another example of external stimulus. \( A(0) \) is the initial static architecture. \( F^* \) is a function that "adds" all stimuli to the architecture change function \( H(a) \). The architecture change function transforms the structures and the relations that together constitute the architecture. \( A(t) \) is the architecture at any given time. There is also a feedback loop denoted \( -1 \) that prohibits the system from unlimited expansion.

A static software architecture is defined in (6). This allows us to take into account that a software architecture can change during development.

\[ \begin{align*}
A(0) \\
1 > t_0
\end{align*} \] (6)

A software system where parts of the system is replaceable or extendable at run-time is considered as having an \( E(t) \) that affects \( A(t) \). From this definition it can be seen that the dynamic software architecture where structures cannot be replaced at run-time is bounded by \( A(t) \), whereas systems where parts can be replaced at run-time is bounded by \( E(t) \).

A static architecture is time invariant in terms of that it does not change by just time going by. It needs external stimuli, while a dynamic architecture may be time variant.

An shorter form of denoting the dynamic properties of the architecture is to use the unit step function \( \delta(t) \) (7) for a dynamic architecture, and (8) for a static architecture. Using the unit step function, it is also possible to discuss the architecture during well-defined temporal spans.

\[ A(t) \delta(t) \] (7)

\[ A(t) \delta(t) \] (8)
An informal description of the given definition would be that the static software architecture is a building plan for the dynamic software architecture.

**Aggregation level.** A non-aggregated structure is considered as an atomic unit. If a structure is considered as being aggregated from one or more structures, it is considered as an aggregated structure. These properties of structures are defined using the Extended Backus-Naur Form (see for example [1]) in (9).

\[ \text{Aggregated Structure} = (AS) \]
\[ \text{Non-Aggregated Structure} = (NAS) \]
\[ (AS) = (AS) \cdot (NAS) + (NAS) \]

(9)

See figure 1 for an example of an instantiation of an aggregated structure.

It is tempting to number aggregation levels using literals. This would however make it difficult to discuss reverse engineering [5] and forward engineering at the same time. During reverse engineering it is possible that non-aggregated components are identified before aggregated components [8]. This would give the non-aggregated components aggregation level 1. During top-down forward engineering, aggregation level 1 would probably be assigned to the highest aggregation level. Therefore, each organization must assign their own name to the aggregation levels. This is done using literal plug-ins.

**Literal plug-ins.** Both the abstraction level and the aggregation level dimensions are open-ended, that is there is no scale applied to them. An applied scale is called a literal plug-in, since a set of literals is applied on each of these two dimensions.

An example of a literal plug-in for the aggregation level dimension is given in (10). An example of a literal plug-in for the abstraction level dimension is given in (11).

\[ \{\text{System, Subsystem, Component, Class}\} \]
\[ \{\text{FunctionalUnit, ConcurrentStateMachine, SoftwareProcess}\} \]

(10)

(11)

The TOPSA does not attempt to prescribe which literal plug-ins should be used, since those are affected by many parameters such as the particular product the organization used to create the product and the process used.

**Additional definitions.** A software architecture can be purely realizable. Such an architecture is defined as all structures being used are on a realizable level. A purely conceptual architecture has not been implemented in code, but as the transformation function \( f = 1 \) is a possibility, the conceptual architecture may be equal to the realizable software architecture.

**2.3 The TOPSA vector**

Using the definitions given, the software architecture description vector for any given time can be defined as in (12), where abstraction level refers to any conceptual level \( C_{dep} \) or a realizable level \( R \). Dynamism is either static \( S \) or dynamic \( D \), and the aggregation level can given at any level, denoted \( A_{agr} \).

\[ \text{TOPSA} = \{\text{AbstractionLevel, Dynamism, AggregationLevel}\} \]

(12)

**3. Discussion**

This section gives some more examples of use of the TOPSA vector. In section 3.1, it is illustrated how the TOPSA can be used to describe understandability of an architecture through describing the distance between a conceptual architecture and a realizable architecture. In section 3.2, the use of the TOPSA in education describing performance properties is shown. Section 3.3 discusses issues related to the representation of architectures. A longer example of the use of the TOPSA is provided in section 4.

**3.1 Understandability properties**

Assuming that a system is easy to understand if its realization architecture reflects real-world structures and dynamics, (13), (14) and (15) should hold true for any \( r \).

\[ \text{RealWorld}(r) = \{C_{dep}, D, A_{agr}\} \Rightarrow \varepsilon \]

(13)

Copyright 1999 Lars Brutschy, Per Rostgaard
\[ [C_{AV}, D, A_{AV}] \rightarrow [R, D, A_{AV}] \rightarrow t \quad \epsilon \rightarrow 0 \]  
\[ [C_{AV}, S, A_{AV}] \rightarrow [R, S, A_{AV}] \rightarrow t \quad \epsilon \rightarrow 0 \]  

(14)  
(15)

Here it has been illustrated how the TOPSA model can aid in transferring a comprehensive description of a software architecture with plausible high understandability properties. The realizational software architecture is closely related to the conceptual software architecture. The conceptual software architecture mirrors the real world at any given time.

3.2 Education

The TOPSA-vector can be used in education. For example, to explain to a student that for performance reasons, the change-state (creation of structures) in a software system must be kept low during rather than the initialized-state, it is possible to state that the system must obey (16).

\[ \frac{\partial}{\partial t} [TOPSA(t)] = [R, D, A_{AV}] \]  

(16)

Fig. 4. Change in state of dynamic architecture vs time

In a performance sensitive software system, real world concepts may not be implemented as good representations of reality. Using the same definitions as above, one may be forced to implement a system that have equations like (17), (18) and (19).

\[ Real\ World(t) - [C_{AV}, D, A_{AV}] \rightarrow \epsilon \quad \epsilon \neq 0 \]  
\[ [C_{AV}, D, A_{AV}] \rightarrow [R, D, A_{AV}] \rightarrow t \quad \epsilon \neq 0 \]  
\[ [C_{AV}, S, A_{AV}] \rightarrow [R, S, A_{AV}] \rightarrow t \quad \epsilon \neq 0 \]  

(17)  
(18)  
(19)

This example illustrates how the TOPSA model can be used to comprehensively describe the architecture systems where implementation differs significantly from analysis models. Examples of such systems are high throughput communication protocols, where an architecture with multiple layers is used for analysis, but during implementation, the layers are compressed into a single layer.

3.3 Representation

The TOPSA model assumes the existence of certain types of architecture, but does neither address the representation of models used to describe the architecture at certain points in TOPSA space, nor does it define particular views of such representations.

Adding representation to the TOPSA vector has been considered, but the idea has been rejected for two main causes:

- Representation is concerned with creating a transferable image of any point in the TOPSA space. The representation does not intrinsically hold any information regarding the properties given in the TOPSA vector.

Copyright 1999 Lars Brantholm, Per Ransnas
4. Software development process

4.1 Introduction

The TOPSA model views any engineering activity as a set of transformations between different points in the TOPSA space. A software development process for an inexperienced organization may explicitly state that a large set of transformations should be performed, while a more experienced organization may use fewer but larger transformation steps.

A software development process - or reverse engineering process - can be seen as a set of transformations. The different transformation steps may be performed using different tools, notations or required inputs/outputs. Yet, they can all be described using the TOPSA model. It is believed that this may be of value to practitioners and researchers wishing to compare different development processes. The formality may also make it easier to measure the outcome from architectural processes. The purpose of section 4 is to illustrate how the TOPSA can be used both to guide designers through different architectural levels, as well as to describe how an architectural process has proceeded.

In section 4.2, the use of the TOPSA vector is demonstrated by describing how the construction of a high-performance distributed embedded system at Ericsson Microwave Systems AB took benefit of the TOPSA vector. The TOPSA vector was used to describe various properties of the architecture. The purpose of using the TOPSA was to guide designers through the development of the system.

The aggregation level and abstraction level literal plug-ins used were (20) and (21). These, together with the dynamism dimension, constitute the space where architecture was studied. This is illustrated in figure 5, where (22) holds true. In (22), tₖ is a point in time, where tₖ occurs before tₙ etc.

\[
\text{System, Component} \quad (20)
\]

\[
(\text{Conceptual Architecture, Realization Architecture}) \quad (21)
\]

\[
t_k < t_{k+1} < \ldots < t_n \quad (22)
\]

![Fig. 5. Project instantiation of TOPSA space](image)

4.2 Example: Use of the TOPSA in an industrial project

In the industrial project, an embedded system including completely new hardware as well as software was to be developed. The software development organization initially lacked deep knowledge of the domain as well as of hard real-time systems. It was known beforehand that CPU-resources would be scarce. Thus, the project was a
challenge from the perspective of creating an easily maintained and extended system without sacrificing performance.

Other quality requirements included that the product would be maintained for at least 15 years, and there would be multiple versions available for different customers, with independent life cycles for different versions. The gain trade possible by architectural thinking in such product family/product line systems is illustrated in for example [N].

The functional requirements were well known and expressed using message sequence charts [17] and a variant of use cases [18] much like [27] that is tailored towards systems that are easily implemented as state machines, such as embedded systems.

4.3 From requirements to conceptual dynamic architecture.

The functional requirements were used to elicit a software architecture with the properties in (23), i.e. a conceptual dynamic architecture at a low aggregation level. See also Figure 5. This is similar to the elicitation of an object-model, with explicit representation of creation and deletion of structures. The notation used was informal. The choice of the conceptual and the dynamic dimension was due to that these two dimensions are close to how humans think according to [25]. Given more time for training, any OOA method/representation could have been used.

\[ T_{OPS/A} = \{C, D, A_{int} \} \] (23)

When the design group was convinced that all conceptual dynamic structures had been incorporated, structures were aggregated (24), i.e. components were grouped together to hide complexity.

\[ (T_{OPS/A}) = \{C, D, A_{int} \} \xrightarrow{s} (T_{OPS/A}) = \{C, D, A_{int} \} \] (24)

By preserving \( Z_i \) later viewing the conceptual architecture at various aggregation levels was supported. The higher aggregation levels were used to explain the software to new designers, whereas the lower abstraction levels were used to ensure that no details in the requirements were overlooked. \( T_{OPS/A} \) was compared to other systems being part of the overall distributed system, described as \( \{C, D, A_{int} \} \). This resolved some interactions-challenges between parts of the distributed system.

In order to further simplify the process of explaining the software architecture, creation/deletion of structures were removed in transformation \( Z_i \) (25), as a conceptual static architecture was designed.

\[ (T_{OPS/A}) = \{C, D, A_{int} \} \xrightarrow{s} (T_{OPS/A}) = \{C, A_{int} \} \] (25)

This transformation did not reveal much information, since \( Z_i \) was close to 1. In this case this implies that most software processes and threads were created at system start-up. In other systems, it is likely that this step would reveal more structures related to initiating, maintaining and removing the structures of the dynamic architecture.

At this point, the software design group was confident that they had understood the domain well. All use cases were compared to \( T_{OPS/A} \), as to ensure that no functionality had been overlooked. As to ensure that the quality requirements were met, all structures in \( T_{OPS/A} \) were assigned a number of parameters: Likelihood to change (denominated through change activation) and performance sensitivity.

During the transformation (26) when the conceptual dynamic architecture was transformed into a realisational dynamic architecture, a real-time system expert was consulted. \( T_{OPS/A} \) balanced the maintainability requirements and the performance requirements by choosing realisational structures in such a way that the performance critical structures in the conceptual architecture were easy to make efficient, while the conceptual structures most likely to change were allocated to separate realisational structures. Mandatory COTS components were taken into account in transformation (26). \( Z_s \) was recorded in detail using a table. A simplified \( Z_s \) is graphically illustrated in figure 6.

\[ (T_{OPS/A}) = \{C, D, A_{int} \} \xrightarrow{s} (T_{OPS/A}) = \{P, D, A_{int} \} \] (26)

The next transformation was (27), where the realisational dynamic architecture was transformed into a realisational static architecture. At this point, system initialization structures were added. \( T_{OPS/A} \) is the actual source code, that is used to build and manipulate \( T_{OPS/A} \). For organizational and project reasons, a higher aggregation level \( T_{OPS/A} \) was created (28). It was used to facilitate division of work between development teams.
Fig. 6: A graphical illustration of $Z_G$. Similarity of concern is represented by spatial proximity in a 3-D space, where the transformation function is the mapping between two TOSA-views. This is an example of how the transformational nature of the TOSA model enables representation of the latent space of the spatial environment [20] in terms of the basic concepts of similarity of concern.

\[
(TOSA_{i}) = (R, D, A_{i}(D)) \quad (TOSA_{i+1}) = (R, S, A_{i+1}(D)) \quad (TOSA_{i+2}) = (R, S, A_{i+2}(D))
\]

4.4 Results

The use of the TOSA in the project has been qualitatively evaluated, and the product has been evaluated in terms of how well it fulfills its functional and quality requirements.

Process evaluation. The use of a conceptual architecture helped to enable the designers to understand the domain. The conceptual architecture was used during discussions with project management, who wished to use it during project management. Since there was a clear and retained transformation between conceptual, functional, and the realization architecture, software designers could give input regarding estimated time as well as need for particular expertise. This was perceived as beneficial by the project management.

The conceptual architecture was used during discussions with product strategists. They could provide information on likely change hotspots, i.e., structures that would be likely to change because of other product line considerations, or particular future customer requirements, or changing hardware. The realization architecture took that information into account and unless performance required it, the change hotspots were made easy to change in the realization architecture on the expense of slightly increased hardware requirements. Thus a "complexity managing software architecture" [21] was created.

Even though the organization had a development process in place, that prescribed the use of multiple views on "the architecture", the TOSA was perceived as beneficial, as the whole defined differences between points in the updated TOSA space. Instantiation made it clear on what architecture was used and how it was being performed. A finding, that cannot entirely be attributed to the TOSA but to the consideration of dynamic architectures before the static architectures, was that this order helped in identifying otherwise easily forgotten initialization code that created and maintained the dynamic parts of the system.

Product evaluation. The product has been evolved from two perspectives. Ability to fulfill the functional requirements and ability to respond to the quality requirements, mainly the ability of the product to be maintained for a long time and the ability to be developed into multiple versions for different markets.

The functional requirements including requirements that challenge performance have been verified and validated by applying the use cases combined with environmental stress conditions. The maintainability requirements were validated through coding using a SAAM-like [19] method; the change scenarios described by the SAAM were identified prior to creating the realization software architecture, and thus could be taken into account beforehand. The change scenarios were based on the business strategy rather than end-user input. The system is now being released in its first version, and as change was anticipated already during the creation of
the software architecture, we are convinced the product will withstand the expected changes. Thus the system is "future-proof," according to [33]. Should the future views request be the same or unexpected changes process described by the change scenario, the realization software architecture may need to change. Since the realization architecture is less analogous to reality than the conceptual architecture, the realization architecture is more likely to change.

The use of TOPSA made the distinction between conceptual architecture and realizational architecture clear, and forced the creation of architecture models suited for different purposes and different stakeholders. As the transformation functions have been retained, product evolution is facilitated as it is clear how models describing different points in TOPSA space are related to each other.

5. Related work

Most work within software architecture has been directed at the TOPSA()] = {I, S, P, A\sub{in}}. Examples of this are many patterns. There are also formalism used for describing components and connectors at the realization level, see for example [12, 20]. Notations are available for describing conceptual architectures, for example [6]. The description of conceptual abstraction levels given in section 2 maps well to the connectionist school in cognitive science [25], where thoughts are modelled as "particle distributed processing". This indicates that conceptual architectures are not to be humans thought.

Less work is available on the process of transformation from the conceptual domain to the realizational domain. One example is the Application Layer Framing (ALF) [10] architecture, that starts off with a conceptual architecture (the OSI reference model, see for example [10]) and ends with a realization architecture that is different from the OSI reference model. Another example of a transformational methodology is the Schwafer & Mellor methodology [28].

Soni et al. [30] offers a model that is related to the TOPSA model. Four architectural categories are described: Conceptual architecture, module interconnection architecture, execution architecture and code architecture. Code architecture describes how the source code etc. are organized in the development environment, and it has no correspondence in the TOPSA model, as it does not describe a software architecture, rather it is a view of the implementation management view on the items used to carry the architecture. [30] does neither explicitly address aggregation, nor are definitions given formally. The conceptual architecture is equal to one particular static conceptual architecture at a high aggregation level. Execution architecture is an example of a dynamic realization architecture. The module interconnection architecture includes both functional decomposition and layers. Functional decomposition can be equated to using aggregation on a static realizational architecture, while layers can be seen as the usage of the layer architectural style on a static conceptual architecture. Kuchlén [22] offers another related model, the "4+1 View Model". The model was made to remedy the problem of representation, while the TOPSA model is more generalized and that allows the discussion of architecture descriptions, such as both [30] and [22]. The latter describes a logical view, a process view, a physical view and a development view. The development view is equal to Soni's code architecture. The logical view equals Soni's conceptual view. The process view is one particular abstraction level in TOPSA terminology, where it seems both static and dynamic properties are considered. The physical view describes the mapping of software onto hardware and reflects its distributed aspects. Thus the physical view encompasses several points in TOPSA space. Both the static and dynamic architecture, but at the realizational level.

A third way of describing architectural structures by views is provided by [4]. The views are well described, as are some possible uses of the views. The view expressed regarding software architecture structure corresponds fairly well to [22], but some extra views are given.

Another related model is [3]. Through a comparison between conventional design and software design, [3] identifies a clear distinction between the static source code implementation and the dynamic system execution, as well as a conceptual model of architecture. The main difference between [3] and the TOPSA is that the TOPSA suggests that an architecture can be described by three orthogonal dimensions, while [3] suggests that there are three different representations: The Conceptual Model Representation, The Static Implementation Representation and the Dynamic Operational Representation(s).

The main difference between the TOPSA model and the other models (3, 4, 22, 30) is that rather than describing and defining different views on architecture, the TOPSA defines orthogonal properties of a software architecture. Thus it is possible to discuss "a conceptual dynamic architecture" at a particular aggregation level, rather than just referring to either an execution architecture or a conceptual architecture or a logical view or a process view. However, there is value of using the views described in [4, 22, 30]:

- They give valuable examples of literal plug-ins on both the abstraction dimension and the aggregation dimension in TOPSA space.
They give examples on views of architectures in the TOPSA-space, that the TOPSA model does not provide. For example, [4] describes a call structure, that could be useful as a view on any point in the TOPSA space.

Thus, the models complement rather than contradict each other.

Apart from related models, other researchers have used some of the vocabulary related to the TOPSA differently. Two important differences [2, 5] are exemplified below.

Allen et al. [2] address the problem of capturing dynamic architectures. They use "dynamic" to denote "systems for which composition of interacting components changes during the course of a single computation". This is distinguished from steady-state behaviour, which refers to "computation performed by a system without reconfiguration". The TOPSA model defines a dynamic architecture differently, as it can be argued that at some point in time, a set of realization-level structures must be created that did not exist prior to execution of the software. In the TOPSA model, steady-state behaviour refers to a system where (29) holds true, i.e., a system where a steady-state as defined by [2] is reached at a certain point in time.

\[
\text{TOPSA}(t) \to [R, D, A_{st}] \to 0, \quad t \to \infty \quad (29)
\]

The expression "properties of architecture" has been used differently by many researchers. For example, [5] defines architectural properties as "concepts related requirements and design decisions that are (or even cannot be) expressed explicitly in what is usually referred to as design", and states "real-time aspects, structural organization and "special quality requirements" such as availability and reliability as examples of architectural properties.

This paper uses the expression "properties of architecture" differently. The purpose is not to describe which requirements an architecture fulfills, rather the purpose is to facilitate discussions regarding an architecture per se. Therefore "properties of architecture" is used differently in the TOPSA context.

6. Summary and Future

The main contribution of this paper is a taxonomy of formally defined orthogonal properties for describing software architectures, that extends the [4] definition of software architectures. The difference between and existence of dynamic and static software architectures is suggested, which extends other related work. Through the increased level of formality, the taxonomy addresses one of the reasons, claimed by [11], for why the definitions of software architecture do not converge: "the foundations [for software architecture] have been imprecise". An industrial project using the TOPSA is presented to illustrate the use of the TOPSA, and several other uses of the TOPSA are described. The usefulness of the TOPSA for discussing related work is demonstrated.

The TOPSA uses various degree of formality, and is not a final model in any sense, e.g., it does not include the explicit notion of neither components, nor connectors and relations between components, though it may be possible to include these in the future.

The TOPSA can facilitate discussions regarding software architecture during development and evolution. We believe that it may be possible to create a new set of metrics based on edit distance [13, 21] that can be used on for example maintainability and understandability, by assessing the transfer function Z from conceptual architecture to realization architecture, and also possible from static architecture to dynamic architecture. Even the existence of different architecture models at various points in TOPSA-space could possibly be used to predict relative understandability metrics, by measuring distance between chosen models in TOPSA-space to the cognitive processes for different types of maintenance by empirically identified by [24].

7. Acknowledgement

This work was partly funded by The Swedish National Board for Industrial and Technical Development (NUTEK), grant 1K1P-97-09690. Prof. Claes Wohlin at the Software Engineering Research Group at the Dept. of Communication Systems and employees at Ericsson Microwave Systems AB have given insightful comments on this paper, as well as participants of the First Nordic Workshop on Software Architecture. The industrial project described was conducted while the author was employed at the Q-Labs Group.
8. References


17. ITU-T, Recommendation 214; Message Sequence Charts (MSC), Telecommunication Standardization Sector of the International Telecommunication Union, 1996.


