International Journal of Computers and Applications

ISSN: 1206-212X (Print) 1925-7074 (Online) Journal homepage: http://www.tandfonline.com/loi/tjca20

Formal Analysis of The Structural Correctness of Slc Descriptions

S.J. Choi & W. Scacchi

To cite this article: S.J. Choi & W. Scacchi (2003) Formal Analysis of The Structural Correctness of Slc Descriptions, International Journal of Computers and Applications, 25:2, 91-97, DOI: 10.1080/1206212X.2003.11441688

To link to this article: https://doi.org/10.1080/1206212X.2003.11441688

International Journal of Computers and Applications, Vol. 25, No. 2, 2003

FORMAL ANALYSIS OF THE STRUCTURAL CORRECTNESS OF SLC DESCRIPTIONS

S.J. Choi* and W. Scacchi**

Abstract

This article presents a way of assuring the correctness of configured software descriptions throughout software lifecycle activities. To achieve this, we draw on concepts in the areas of software architecture and software engineering environments to provide a formal and automated basis for achieving verification and validation of software concepts, for assuring the structural correctness of software lifecycle object configurations based on the objects' resource attributes and resource relations and the transformation of these attributes and relations throughout the software lifecycle. We formalize these concepts such that we provide a set of lemmas and closure theorems that substantiate our concept of software lifecycle correctness. All proofs are also included. These concepts in turn provide the basis for an integrated environment with automated tools that can ensure the correctness of configured software descriptions. Thus, through our approach and formalisms, we demonstrate a way to combine software verification and validation techniques with software architectural definition concepts that can support a software engineering environment.

Key Words

Structural correctness, V&V, architectural definition, formal analysis, SEE

1. Introduction

Determining the correctness of a software system can involve verifying that the implementation conforms to its specification and design, whereas validating the implementation satisfies its requirements. *Formal verification* usually denotes that it is possible to rigorously show that a formal software specification can be consistently and completely transformed into a formal design, and in turn into an operational source code form [1]. Software lifecycle verification seeks to determine the degree to which the products from a given development phase or activity fulfills the specifications established during prior lifecycle phases. *Validation* often denotes that each operational requirement that can be systematically traced to some software functionality or condition is satisfactorily realized in the source code implementation. Formal verification offers the intrinsic value of a rigourous analysis that shows how a source code program can be derived from its formal specification. However, in practice, formal verification is often limited to small or modest-size programs with well-defined requirements and relatively stable specifications [2].

The objective in this article is to describe a formal approach to verifying and validating the development of large software systems throughout their lifecycle. Our notion of the correctness of a software system throughout its lifecycle is limited to the *structural* formalism we utilize, and thus we make no claim about proving *behavioural* properties of programs correct. This is similar to the distinction between the development of a program that can be "compiled" (syntax, type, and usage checked) from one whose behaviour can be "executed" at runtime. In this regard, a program that cannot be successfully and correctly compiled cannot be executed. Accordingly, our view is that a set of software lifecycle descriptions that cannot be formally shown to be structurally correct, as we will describe, cannot be behaviourally correct. But we believe our approach is practical and can be applied to software development efforts that involve many people who may be distributed across multiple sites in a loosely couple manner [3, 4].

There are two principal concepts embodied in our approach, and an associated environment that supports their operationalization. The first concept involves the generalization, extension, and application of *architectural definition language* (ADL) concepts [5], to all software lifecycle descriptions, not just to software architecture design activities. ADLs address the structural organization of components of a system, their interrelationships, and principles and guidelines governing their design and evolution over time [5]. Furthermore, recent advances with ADLs show how they can be: correctly refined from abstract to lower level forms or programs [6, 7]; formally defined and logically analysed [8]; defined for comparing different architecture patterns [9]; and defined for specifying

* Computer Science Department, California State University Fullerton, Fullerton, CA 92834-9480, USA; e-mail: sjchoi@ecs.fullerton.edu

** Information and Computer Science Department, University of California, Irvine, CA 92697-3425, USA; e-mail: wscacchi@ics.uci.edu

(paper no. 202-1129)

behavioural architectural models incorporating concurrency, synchronization, dataflow, and timing properties [10]. Our effort involves development of a formalism that views all objects created as a software system's lifecycle components (individual requirements, design diagrams, test plans, etc.), each with an *interface* through which information resources are provided to or required by other lifecycle descriptions. Resource type and configuration relations are then associated with the interfaces, which must be compatible for two or more software objects to be interconnected. As such, software lifecycle components can be composed or interconnected into lifecycle stage descriptions (e.g., requirements analysis, design, testing). Further, descriptions associated with successive software lifecycle stages are also maintained as *neighbour* relations, which denote ordered mappings of interdescription transformations. Thus, in this regard, we have adapted concepts from ADLs to interrelated software system descriptions of mixed notation type and formalization.[2] This formalism also serves to provide a basis for the second concept. We define and formalize the structural correctness of configured software descriptions (i.e., a software system's lifecycle descriptions) in terms of their consistency, completeness, and traceability. The formalism leads to lemmas and closure theorems (CTs) that form the basis for supporting the structural correctness. Last, we briefly describe a software engineering environment that supports the construction and incremental analysis of the structural correctness of the software object descriptions in configured groupings.

2. Definitions and Background Concepts

The development of large software systems is comprised of a set of activities known as the software development process or *software lifecycle*. Various names and forms of these software lifecycle activities have been identified by many authors. Some prefer to group these activities into standardized phases with well-defined products, and others only see an arbitrary number of intermediate transformations that progressively take user requirements into source code. In our approach, the number of stages depends only on the choice of which intermediate lifecycle products are intended to persist, and whether there is a formal language notation associated with the product of each development stage. To focus our discussion, we adopt the following software lifecycle stages: requirements, functional specification, design, implementation, and testing and maintenance. We note that these software lifecycle activities suggest many possible relationships among the various lifecycle descriptions. However, they do not necessarily imply a unique sequential order of software development. Further, we assume that functional specification, design, and implementation will be described with language notations, and the others employ informal descriptions. The basic concepts utilized in this work—descriptions, objects, predefined attributes of objects, and resources—can be found in [11]; the following sections briefly describe the essential background for understanding our concepts.

[2] As should be clear, we are adopting some concepts from ADLs, but at this point we are not addressing advances of behavioural architectural concepts [16], nor architectural styles [23].

2.1 Relationships

Two objects occurring in the same description can have two types of resource relationships. They can be *require-related*, meaning that one object imports a resource from the other object, or they can be *provide-related*, meaning that one object exports a resource to another object. Clearly, these two relations are the inverse of each other. The rationale to include both in our model relates to the closure properties that are central to our definition and formal analysis of structural correctness, as defined later.

Part-of is a simple relation between a basic and a composed object or between two composed objects occurring in the same description. The relation identifies a hierarchical membership relationship between two objects and is useful in identifying the components of a composed object.

Inter-related is the only relation in our model that relates objects from two different descriptions, signifying that these two objects are related through a *resource transformation* (RT). The concept of RT is discussed in more detail below.

2.2 Neighbours and Proper Neighbours

A complete system is a partially ordered hierarchical composition of all of its included lifecycle descriptions. Each lifecycle product description corresponds to a subhierarchy of the system lifecycle hierarchy. In a top-rooted hierarchy, the first lifecycle product (requirements) is the left-most subhierarchy. Intermediate lifecycle products are then ordered left to right according their appearance in the lifecycle model. In this way, design and implementation descriptions, for example, represent subhierarchies of objects that are adjacent to each other. An object y is a *neighbour* of object x if y has an inter-related relationship with x. If y is an object of an immediate previous description (i.e., is located in an adjacent subhierarchy to the left), then y is *left-neighbour* of x. Similarly, if y is an object of an immediate following description (in an adjacent subhierarchy to the right), then y is *right-neighbour* of x. Furthermore, we define that a set S of objects is a *proper neighbour* of an object x if all elements of S are all right-neighbours (left-neighbours) of x and x is the only left-neighbour (right-neighbour) of all objects in S.

2.3 The Concept of Resource Transformation

In general terms, an RT deals with how resources provided by an object in one description are *transformed* into resources provided by objects in a neighbouring description. Notice that an object's required resources are provided by some other objects; thus, we need to consider only the provided resources. The specific realization of each RT is not the focus of this study. Nonetheless, intuitively, the realization of RTs can be viewed as the

behavioural content that is added by developers in evolving the description of a software object description into its neighbours(s), whether forward or backward in the lifecycle [12, 13]. This may be achieved through description creation, refinement, and editing in one or more notations. Alternatively, it may be realized automatically when employing an automated transformation refinement or implementation system. For our approach, what differs between the two is only the determination of which transformed descriptions will persist, be evolved, or be treated as deliverable products. Given this concept of RT, we distinguish two types of transformations as the minimum relevant to our analysis.

Expanding-transformation (*1-to-*n *transformation*). This type of transformation signifies that a resource provided by an object is transformed into one or more resources provided by the neighbouring objects. Expanding-transformation occurs if all neighbouring objects are properly transformed, as is explained next.

Merging-transformation (n-*to-1 transformation, for* n *>1*). This type of transformation signifies that two or more resources are transformed into a single resource. The merging-transformation can occur, for example, if two objects have the same neighbour. This type of transformation is also called the *generalized transformation.*[3]

3. Formal Analysis

In this section we formalize the concepts developed in the previous section by asserting three postulates and proving several lemmas that lead to three CTs. The main lemmas included in the following sections are two *resources lemmas*, the *attributes lemma*, and two *completeness lemmas*. These lemmas lead to the immediate proof of the corresponding CT. Practical examples of the lemmas and theorems are described in [11].

3.1 Postulates

In dealing with the development of software systems, there are many known heuristics or facts derived from experience that are very difficult, if not impossible, to prove or include as part of a formal system. In formalizing the concepts of structural correctness, we encounter these type of problems, and our approach to dealing with them has been to postulate their existence. Accordingly, at the beginning of formalization work, we introduce three postulates that are required to make our formalization complete. The first postulate relates to the fact that if an object provides a resource in a lifecycle description, then its neighbours must provide the corresponding transformed resources.

Postulate 1 [Existence of the Transformed Resources]. Let r be a resource provided by an object x and S be proper neighbours of x. Then S must provide a non-empty set of resources R where R is the transformation of r.

The second postulate relates to the fact that if an object x requires a resource from another object y in a given life-cycle description, then the neighbours of x require at least one resource from the neighbours of y.

Postulate 2 [Existence of the Required Resources]. Let x be an object that requires a resource r from another object y in a lifecycle description, S1 the proper neighbours of x and S2 the (proper) neighbours of y. Then there exists a non-empty set of resources R, the transformation of r, and a resource R$i \in$ R such that S1 requires Ri from S2.

The last postulate relates to the fact that there are "invisible resources"; this means there exist some resources in the neighbours of an object that are required (consumed) within the neighbours itself.

Postulate 3 [Existence of the Invisible Resources]. Let S be the proper neighbours of an object x. Then there may exist a resource r provided by an object Y$i \in$ S such that r is required Y$j \in$ S and $i \neq j$.

Lemma 1. Let x be an object and S be the set of all its right-neighbours. Let x and S be proper neighbours. Then all of the resources provided by x must be transformed into resources provided by S.

Proof. Assume that x provides a set of resources $r1, r2, \ldots, rn$. As S is proper neighbour of x, then by Postulate 1, S provides the resources R1, R2, ..., Rn where Ri is a non-empty set and the respective transformation of resource ri. Hence, all provided resources of x must be transformed into provided resources of S. □

Resources-Lemma 1. Let x and y be require-related, and let the set of objects S1 be the proper right-neighbours of x and S2 all right-neighbours of y. Then there must be at least one object in S1 that has require-related relation with an object in S2.

Proof. Let r be a resource required by x from y. Then by Postulate 2, there exists a non-empty set of resources R, the transformed resources of r, and a resource R$i \in$ R such that S1 requires it from S2. Hence, there must be an object X $\in$ S1 that requires Ri provided by S2. As RT Ri must be provided by an object Y $\in$ S2, then there exists an object X $\in$ S1 that is require-related to an object Y $\in$ S2. □

Resources-Lemma 2. Let x and y be provide-related, and the set of objects S1 be the proper neighbour of x and S2 be all right-neighbours of y. Then there must be at least one object in S1 which has a provide-related relation with an object in S2.

Proof. Let r be a resource provided by x to y. Then, by Postulate 1, S1 must provide a non-empty set of

[3] In this work we distinguish between proper transformation and generalized transformation only in those cases where the distinction is strictly required in the context under discussion. Otherwise, transformation refers to proper transformation.

resources R transformed resources from r. Now, by Postulate 2, there must exist a resource R$i \in$ R such that S2 requires Ri from S1. As such, there must be an object X $\in$ S1 that provides Ri to Y $\in$ S2. Hence, there exists an object X $\in$ S1 that is provide-related with an object Y $\in$ S2. □

The Resources-Attributes-Lemma. Let S1 be the set of all objects that are provide-related with an object x, and let S3 be the set of all right-neighbours of S1. Let S2 and x be proper right-neighbours. Then all require-related relations of objects in S2 must be contained in S2 $\cup$ S3.

Proof. Assume that $y1, y2, \ldots, yn$ are objects that provide resources to x (S1 is provide-related to x). Without loss of generality, assume that $y1$ provides resources $r1, r2, \ldots, rm$ to x and that Y1 is the neighbour of $y1$. Then by Resources-Lemma 1, there must be at least one required-related relation from S2 to Y1. By definition of RT, Y1 must provide the transformed resources of $r1, r2, \ldots, rm$. Therefore, all require-related relation of S2 must be contained in Y2, Y3, ..., Yn, the respective neighbours of $y2, y3, \ldots, yn$. As Y1, Y2, ..., Yn are contained in S3, all the require-related relations of S2 on the resources $r1, r2, \ldots, rn$ are contained in S3. Furthermore, by Postulate 3, any object z$i \in$ S2 can require resources from another object z$j \in$ S2 with $i \neq j$. Hence, all require-related relations of S2 are contained in S2 $\cup$ S3. □

Resources-Completeness-Lemma 1. Let n be the number of resources required by an object x, and let S be the proper neighbours of x. Then S must require at least n resources.

Proof. The proof follows directly from Postulate 2. Assume that $r1, r2, \ldots, rn$ are all of the resources required by object x and that R1, R2, ..., Rn are the respective transformations. As S is the proper neighbour of x, by Postulate 2, S must require at least one resource from each Ri. Hence the total number of resources required by S must be equal to or larger than n. □

Resources-Completeness-Lemma 2. Let n be the number of resources provided by an object x, and let S be the proper neighbours of x. Then S must provide at least n resources.

Proof. Assume that $r1, r2, \ldots, rn$ are all of the resources provided by object x, and that for all $i \neq j$, $ri \neq rj$, that is, there are in total n different resources r. As S is proper neighbour of x, x has an inter-related relation only with S, and thus by Lemma 1 all of the resources required by x must be transformed into resources provided by S. Call these resources R1, R2, ..., Rn. As Ri is a non-empty set, Ri must include at least one element in it; the cardinality of set Ri is equal to or larger than 1 for all i between 1 and n. Therefore, the cardinality of the set formed by the union of all the Ri's, for i from 1 to n, must be equal or larger than n. □

4. The Concept of Structural Correctness

Structural correctness concerns how configured software descriptions conform to preceding neighbour descriptions and to the software system requirements. In our view, a software system is structurally correct if all of its lifecycle descriptions are *structurally traceable*, *consistent*, and *complete*. Clearly, structural correctness deals with all software lifecycle descriptions, as we are able to trace the descriptions and transformations of all of their objects and resources in a consistent and complete manner. In the following paragraphs, we discuss completeness, consistency, and traceability and introduce the *correctness constraints* (CC), which are important concepts in analysing software lifecycle correctness. In addition, we will highlight the main formal results by calling them CTs. These theorems are instrumental in analysing the presence of inconsistencies and *incompleteness* that may occur between two different descriptions. By checking these theorems, it is possible to determine whether a description of an activity has been consistently and completely transformed into the description of a different inter-related activity.

4.1 Structural Traceability

Structural traceability shows to what extent all resources in a description can be identified (internal traceability) and that objects have left- and right-neighbours (external traceability). *Internal traceability* is primarily concerned with identifying resources within a description and is defined in terms of the following two constraints.

- *provide-traceability*: All resources provided by a description should be used.
- *require-traceability*: All resources required by a description must exist.

These constraints are applicable to all software descriptions. Resources that are used in a description must be present, and all resources that are provided in a description should be used within that description. In the requirements description, for example, if an object called "Section-1.1" references an object called "Section-1.2.1" then it is necessary that the object called "Section-1/2/1" exist. Application of these constraints to other descriptions is straightforward. *External traceability* is primarily concerned with identifying the objects occurring in other lifecycle activities and is defined in terms of the following constraint.

- *neighbour-traceability*: All objects must have left- and right-neighbours, except the objects belonging to the left-most (requirements) and right-most (maintenance) descriptions. The objects in the requirements description do not have left-neighbours and the objects in the maintenance description do not have right-neighbours. Neighbour-traceability signifies that each object in the description must be traceable to the objects in the requirements description. If there is an object that cannot be traced to the objects in

the requirements description, then the user must be informed in order to correct this traceability mismatch.

4.2 Structural Consistency

A description is *structurally consistent* to the extent that the provision and use of resources do not conflict within that description (internal consistency) nor between neighbouring descriptions (external consistency). *Internal consistency* occurs within a description and is defined in terms of the following two constraints:

- *intra-name-consistency*: All resources must have unique names within a particular description.
- *use-consistency*: All resources should be used consistently. For example, if an object is defined as a function then it should not be used as a variable. Another example: a resource cannot be included in the provides and requires attributes of the same object simultaneously.

  Use-consistency deals with whether the resources within a description are used *properly* or not. The proper use of resources is only meaningful if the resources have semantics that can be specified formally. As we currently utilize formal languages such as Gist for the specification description, NuMil for the design description, and C for the implementation description, use-consistency is applicable only for these descriptions. However, it should be noted that if we can specify other lifecycle descriptions in a formal manner, we can also apply this constraint in those descriptions.

  External consistency is primarily concerned with consistencies occurring between neighbouring descriptions. It is defined in terms of the following three constraints;
- *inter-name-consistency*: All objects must have a unique name throughout the lifecycle.
- *relation-consistency*: The relations of objects in one description must be preserved in other descriptions. In this context, we introduce the Resources Theorem.

The Resources-Theorem. Let S be all the objects in a lifecycle description. Then all resource relations, provide-related and require-related, in S must be closed.

Proof. Directly from Resources-Lemma 2 (Resources-Lemma 1), if $x \in S$ and $y \in S$ are provide-related (require-related), then at least one object in S1, the sets of all right-neighbours of x, has a provide-related (require-related) relation with an object in S2, the set of all right-neighbours of y. □

- *attribute-consistency*: The attributes of an object in one description must be preserved in other descriptions. In this context, we introduce the Resource-Attribute-Theorem.

The Resource-Attribute-Theorem. Let S be all the objects in a lifecycle description. Then all resources in the requires attribute of the objects in S must be closed.

Proof. Directly from the attributes-lemma, if $S1 \in S$ is the set of all objects that are provide-related with object $x \in S$ (all required resources of x are provided by S), then all require-related relations of objects in S2, the proper right-neighbours of x, must be in S3, where S3 is the set of all right-neighbours of S1. □

4.3 Structural Completeness

A description is *structurally complete* if all resources occurring in a description are traceable to transformed resources of the next lifecycle description. The concept of completeness is formally defined in terms of transformation-completeness.

- *transformation-completeness*: All provided and required resources of an object in a description must be transformed into an equivalent set of resources in a different description. In this context, we introduce the Resource-Completeness-Theorem.

The Resource-Completeness-Theorem. Let S be all the objects in a lifecycle description. Then the number of provided and required resources of the objects in S must be closed.

Proof. Directly from the Completeness-Lemma 1 (Completeness-Lemma 2), if the total number resources required (provided) by an object $x \in S$ is n, then the total number of resources required (provided) by S1, the proper neighbour of x, is at least n. □

In summary, require-traceability, provide-traceability, neighbour-traceability, inter-name-consistency, and intra-name-consistency, must be present in all descriptions. Similarly, all resources used in a description must be traceable within that description or, equivalently, all objects must be traceable throughout the descriptions. However, transformation-completeness, use-consistency, attribute-consistency, and relation-consistency are constraints that are only meaningful within/between formalizable descriptions such specification, design, and implementation descriptions.

Fig. 1 below summarizes our current results by showing the CC in terms of lifecycle descriptions.

	Require	Spec.	Design	Impl.	Test	Maint.
Require-Traceability	x	x	x	x	x	x
Provide-Traceability	x	x	x	x	x	x
Neighbour-Traceability	x	x	x	x	x	x
Intra-Name-Consistency	x	x	x	x	x	x
Inter-Name-Consistency	x	x	x	x	x	x
Relation-Consistency		x	x	x		
Attribute-Consistency		x	x	x		
Use-Consistency		x	x	x		
Transformation-Completeness		x	x	x		

Figure 1. CCs for SLC.

5. Tools Supporting the Correctness Concept

In this section, we provide a brief overview of an software engineering environment supporting the structural correctness called SOFTMAN. A more complete discussion of SOFTMAN may be found elsewhere [12, 13]. SOFTMAN provides a set of necessary tools and integrates these tools in such way that they are accessible through a common user interface called the NSDI (Narrative-Specification-Design-Implementation) Editing Environment.

The tools of the SOFTMAN environment for validating and verifying the software descriptions are:

- *language-directed editing environment*, which provides mechanisms to check these constraints interactively and incrementally, so that violation of CC can be detected at the time of description creation or modification;
- *language processors*, which analyses the CC in batch mode so that the analysis can be done for much larger software descriptions than may be practical with interactive analysis;
- an *object management facility* to maintain the description objects, the relations, and constraints of objects, and a *correctness query processor*, which checks, tracks, and maintains the state of the CC of the objects.

It should be noted that the CC on the descriptions are performed on demand by a SOFTMAN user. In this way, a user (an individual or software development team) may choose to allow inconsistencies, untraceable objects or resources, or incompleteness in the software descriptions they are developing to occur while utilizing the tools. However, a user can request that any or all of the CC be checked, and the tools will report the state of correctness at any time.

6. Comparison to Related Research

The basic idea behind our approach builds upon the idea of "well-formed system compositions" introduced by Haberman and Perry [14] and Tichy [15], and later used by others in formalizing ADLs [8]. These efforts define a well-formed system as the composition of source code program modules whose interfaces are syntactically and structurally consistent. Thus, these notions of well-formed composition correspond to what we have called internal correctness. Similarly, recent advances addressing notations and formal analysis of software requirements descriptions indicate that internal consistency, completeness, and correctness can now be determined [2, 16]. However, these results of these important efforts were directed to single-stage software descriptions, such as software requirements descriptions or source code program (implementation) descriptions. What is new in our work is the extension to support of external correctness of inter related software descriptions, as well as internal correctness of individual software descriptions. In a sense, this means that we have generalized the idea of well-formed composition of system architecture to a diverse range of lifecycle descriptions that can be assembled in large software development efforts. As already noted, these lifecycle descriptions are often described with different notations or formalisms, including informal narrative descriptions. In this way, the number and diversity of descriptions that constitute the set of lifecycle descriptions is open, and not fixed to a specific lifecycle process model, approach, or development paradigm. This leads us to a basis for verifying and validating the structural correctness of a set of inter-related software lifecycle descriptions. We observe that the work cited above led to the development of environments for maintaining the well-formed configuration of software source code in their many related versions. In our work, we too have developed an environment that supports our approach. This means that we can provide basic services for managing the well-formed configurations of the various software lifecycle descriptions individually as well as collectively [15]. Further, we have incorporated the necessary mechanisms to check the CC needed for software lifecycle verification and validation. In this regard, we have developed an approach and supporting tools that combine software architecture and configuration management concepts together with software lifecycle verification and validation services. Our formal analysis and automated support of structural correctness are clearly limited, as weak methods that strive to accommodate architectural refinement, incremental development, and partiality in a manner akin to that advocated by software architecture researchers [6, 7] and by Jackson and Wing in their advocacy of "lightweight" formal methods [17]. However, techniques for specifying and analysing a growing range of formal behaviourally based software lifecycle descriptions, now including requirements and architectural design notations [2, 6, 7, 9, 10], will enable more comprehensive correctness assurance than we have described here. Nonetheless, we believe that our results represent a contribution that builds on the formal, conceptual, and environment engineering work that we have cited in a way that can open up further investigations in each of these areas. One such area for further exploration could address the structural and behavioural definition, formal analysis, and automated support of inter-related software lifecycle descriptions.

7. Conclusion

We have presented the concepts and integrated environment for assuring the structural correctness of configured software lifecycle descriptions. We identified and applied concepts appearing in software architecture definition languages to a diverse range of software lifecycle descriptions that use informal or formal notations. This enabled identification and formal analysis of properties, lemmas, and theorems that define a minimal set of relations, attributes, transformations, and constraints that can be incrementally or fully checked to verify and validate inter-related software lifecycle descriptions.

Initially, we formalized the structural correctness of a software lifecycle configuration in terms of the consistency, completeness, and traceability of the resource relations and resource attributes of its composed software objects, as well as the transformation of these relations and constraints. We then identified the nine CCs and three correctness theorems for the resource relations and attributes. Next, we identified how these constraints can be applied to support the verification and validation configured software descriptions throughout the software lifecycle. However, it should be noted that the CC do not check whether the narrative requirements of a software system are feasible or satisfy user requirements. Thus, on the one hand, a software system conforming to the CC can be said to be well engineered or well configured, but not necessarily behaviourally correct. On the other hand, for a system to be trustworthy and reliable, it must first be shown to be structurally correct.

We also briefly described an integrated software lifecycle engineering environment with a necessary set of tools to support the concept of structural correctness.

Acknowledgements

This article is an expanded version of [11].

References

[1] S. Gerhart, D. Craigen, & T. Ralston, Experience with formal methods in critical systems, *IEEE Software*, *11*(1), 1994, 21–28.

[2] C. Heitmeyer, R. Jeffords, & B. Labaw, Automated consistency checking of requirements specifications, *ACM Transactions of Software Engineering and Methodology*, *5*(3), 1996, 231–261.

[3] C. Loftus *et al.*, *Distributed software engineering* (Englewood Cliffs, NJ: Prentice-Hall, 1995).

[4] J. Noll & W. Scacchi, Repository support for virtual software enterprises, *Proc. of the California Software Symp.*, Los Angeles, CA, 1996, 120–134.

[5] D. Garlan & D. Perry, Introduction to the special issue on software architecture, *IEEE Transactions on Software Engineering*, *21*(4), 1995, 269–274.

[6] R. Gruia-Catalin & D. Wilcox, Architecture-directed refinement, *IEEE Transactions on Software Engineering*, *20*(40), 1994, 239–259.

[7] M. Marconi, X. Qian, & R. Riemenschneider, Correct architecture refinement, *IEEE Transactions on Software Engineering*, *21*(4), 1995, 356–372.

[8] T. Dean & J.R. Cody, A syntactic theory of software architecture, *IEEE Transactions on Software Engineering*, *21*(4), 1995, 302–313.

[9] M. Shaw & R. Deline, Abstractions for software architecture and tools to support them, *IEEE Transactions on Software Engineering*, *21*(4), 1995, 314–335.

[10] D. Lukham & J. Kennedy, Specification and analysis of system architecture using RAPIDE, *IEEE Transactions on Software Engineering*, *21*(4), 1995, 336–355.

[11] S. Choi & W. Scacchi, Formalization and tools supporting the structural correctness of SLC descriptions, *Proc. of the IASTED Conf. on Software Engineering*, Las Vegas, NV, 1998, 27–34.

[12] S. Choi & W. Scacchi, Extracting and restructuring the design of large systems, *IEEE Software*, *7*(1), 1990, 66–73.

[13] S. Choi & W. Scacchi, SOFTMAN: An environment for forward and reverse computer-aided software engineering, *Information and Software Technology*, *33*(9), 1991, 664–674.

[14] A. Haberman & D. Perry, System composition and version control for ADA, *Software Engineering Environments*, Technical Report, Carnegie-Mellon University, Pittsburgh, PA, 1980, 331–343.

[15] W. Tichy, *Configuration management: Trends in software*, vol. 2 (Chichester, UK: John Wiley & Sons, 1994).

[16] M. Heimdahl & N. Leveson, Completeness and consistency in hierarchical state-based requirements, *IEEE Transactions on Software Engineering*, *22*(6), 1996, 363–377.

[17] D. Jackson & J. Wing, Lightweight formal methods, *Computer*, *29*(4), 1996, 21–22.

Biographies

Song-James Choi is an Associate Professor of Computer Science at the California State University at Fullerton (CSUF). He received his M.Sc. and Ph.D. in computer science from the University of Southern California, with emphasis in software engineering. His interests are software engineering, software development process modelling, software acquisition, configuration management, and reverse software engineering.

Walt Scacchi is a research computer scientist at the Institute for Software Research (ISR) at the University of California, Irvine. He joined ISR in 1999 after serving on the faculty at the University of Southern California since 1981. He received his Ph.D. in information and computer science from the University of California, Irvine, in 1981. From 1981 to 1991 he directed the USC System Factory Project, and from 1993 to 1998 he directed the USC ATRIUM Laboratory. His interests include organizational studies of software development, software process engineering, software systems acquisition, electronic commerce, and collaborative work environments. He has published more than 100 research papers and consults widely to clients in industry and government agencies.