A Three-Dimensional Model for System Design Evolution

Amir Tomer\textsuperscript{1} and Stephen R. Schach\textsuperscript{2, *}

\textsuperscript{1}RAFAEL Ltd., Haifa, Israel

\textsuperscript{2}Department of Electrical Engineering and Computer Science, Vanderbilt University, Station B, Box 351679, Nashville, TN 37235-1679

Received 7 November 2001; Accepted 6 March 2002

ABSTRACT

We represent the life cycle of the design of a system in a three-dimensional space with engineering, reengineering, and reuse axes. The three-dimensional model is evolution-oriented. It incorporates not only the evolution that occurs after the product has been produced and delivered, but also three types of system design evolution that take place before the product is produced. Associated with each axis are a mathematical operator and its inverse. These operators, together with their inverses, can describe the various systems engineering activities. The model can be used to describe the life cycle of a product line, the evolution of an individual product within that product line, and even the evolution of an individual artifact. The model can be used in conjunction with any life-cycle model and any set of artifacts. © 2002 Wiley Periodicals, Inc. Syst Eng 5, 264–273, 2002

Key words: evolution; three-dimensional model; engineering; reengineering; reuse

1. INTRODUCTION

Building complex systems requires large-scale and intensive engineering, usually performed by professionals from various scientific and engineering disciplines. According to the International Council on Systems Engineering, systems engineering is “an interdisciplinary approach and means to enable the realization of successful systems” [INCOSE, 1999]. Systems engineering spans a wide range of responsibilities, starting with the early activities of defining the client’s needs...
and requirements until they have been fully implemented in the constructed system. Subsequently, after the system has been delivered to its client, following successful acceptance tests, it tends to undergo extensive modifications throughout its entire life cycle, as a result of removing discrepancies and defects, or the need for improvements or adaptations.

The systems engineering process is described and documented in various guides and standards, of which the two major ones are [ANSI/EIA, 1999] and [IEEE, 1999]. We have chosen IEEE 1220 [IEEE, 1999] as a reference for the concepts and terms introduced and discussed in this paper, but these concepts and terms may also be mapped onto those of EIA 632 [ANSI/EIA, 1999].

According to IEEE 1220 [IEEE, 1999], a typical system life cycle is divided into two stages: development and operations. Although systems engineering is applied to both stages, the “engineering” activities are more commonly associated with “analysis and design,” which, in fact, comprise the development stage. Various activities at different levels (for example, system, subsystem, component, etc.) are performed during development, resulting in a number of work products included in the “integrated data package” [IEEE, 1999: Table 1, 13]. All the architecture and design information of the system is documented in the various elements included in the data package. In the following, we will use the term “[design] artifact” for any such work product. Artifacts may be either basic or compound; a compound artifact is an artifact composed of other artifacts, either basic or compound. Once the design is stabilized, production may start, based on a given configuration baseline [IEEE, 1999: Definition 3.1.4]. Nevertheless, during the operations stage, defects may be revealed and improvements may be called for. In these cases, the engineers go back to the development stage, “reengineering” the system [IEEE, 1999: Definition 3.1.32]. The result of the reengineering process is a new configuration baseline ready for production.

Theoretically, when a new configuration baseline has been established, the previous ones become obsolete and may be discarded. However, many companies nowadays are optimizing their research and development efforts by reusing existing components in creating new products for new clients. Because reusing blackbox components is usually too restrictive, design artifacts at higher levels are extracted, modified as needed, and then reintegrated into the modified architecture. Although typical configuration management (CM) methods and tools are able to keep track of the entire configuration tree and its history, they are far from being sufficient for the purpose of constructing a new product from artifacts located anywhere in the CM system. The cure starts, in our opinion, with the correct modeling of the entire life cycle evolution of the design, over all the products in the same family.

In this paper, we show that there are three directions of system development activities: engineering (analysis and design), reengineering, and reuse. Associated with each activity are a mathematical operator and its inverse. We show that the various activities of systems engineering can be described in terms of the resulting three-dimensional model, a model for system design evolution.

One-, two-, and three-dimensional system development are discussed in Sections 2, 3, and 4, respectively. The three basic operators of system development are discussed in Section 5. We discuss our work in Section 6. Conclusions and future work are presented in Section 7.

2. ONE-DIMENSIONAL SYSTEM DEVELOPMENT

Typical systems engineering is essentially one-dimensional: The activities before production constitute development, and activities after production are regarded as customer support [IEEE, 1999: Fig. 7]. The system engineering process applies throughout the system life cycle to all activities associated with product development [IEEE, 1999]. For the sake of consistency, we will use the principal activities defined in IEEE 1220 [IEEE, 1999], namely:

- Requirements analysis
- Functional analysis
- Synthesis (design).

Because software needs no production (in the engineering sense), software programming is usually regarded as part of development. Therefore, when software-intensive systems are considered, an additional activity should be introduced, as follows:

- Coding.

It should be noted that the coding activity applies at only the component level.

During each of the above activities, artifacts are produced. However, because the IEEE 1220-1998 activity names may be interpreted differently in the general systems context as opposed to the software systems context, in this paper we use a common alternate terminology for the activities. Table I presents the engineering activities and their associated typical artifacts in both contexts [IEEE, 1999; IEEE/EIA, 1997].
In order to keep our model as general as possible, we will refer to the artifacts of both disciplines simply as engineering activity artifacts, using the following abbreviations: RA denotes requirements analysis artifacts, AA, architecture artifacts, DA, design artifacts, and CA, code artifacts. Artifacts of any kind will be denoted graphically as a rectangle with a folded corner, as shown in Figure 1.

Figure 1 also reflects the fact that the engineering process is repeated at the lower levels, that is, at the subsystem level, at the component level, and possibly even at lower unit levels, according to system size and complexity. However, without loss of generality, we will present the model, from Section 3 on, at the system level only. The details may easily be generalized to lower levels, as needed.

The once-through development strategy [IEEE/EIA, 1997: 86] is the archetypal classical life-cycle model for one-dimensional software production. As shown in Figure 1, the engineering team starts from scratch (φ), draws up requirements analysis artifacts, architecture artifacts, and design artifacts. These activities are iterated at the system, subsystem, and component levels, ending up with the coding artifacts of the software components. The collection of code artifacts at a given level is considered as the code artifact of the next higher level.

At this point, the system design is complete, and its elements are ready for fabrication, assembly, integration, and testing (FAIT). FAIT activities are not mentioned in the systems engineering process (SEP) defined by IEEE 1220 [IEEE, 1999: Section 4.1]. We adopt this approach in our model in order to keep the top-down refinement nature of this engineering process. Furthermore, the first three components of FAIT (namely, fabrication, assembly, and integration) produce artifacts, whereas testing does not. Testing is part of the verification and validation that accompanies development for the purpose of approving the artifacts, and therefore is not considered to be an activity in our context. Assembly and integration are certainly activities that produce artifacts, but these are not new artifacts (but rather implementations of existing artifacts); assembly and integration are therefore also beyond the scope of our discussion. As for fabrication, in software it is directly associated with coding, one of the activities in our model. However, parts fabricated in hardware are specific implementations of the design, and thus cannot be considered as design artifacts, and therefore are, once again, beyond our scope.

<table>
<thead>
<tr>
<th>Engineering Activity</th>
<th>Systems Engineering Activities</th>
<th>Systems Engineering Artifacts</th>
<th>Software Engineering Activities</th>
<th>Software Engineering Artifacts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirements analysis (R)</td>
<td>Requirements analysis</td>
<td>Requirements baseline and specification</td>
<td>Software requirements analysis</td>
<td>Software requirements description and specification</td>
</tr>
<tr>
<td>Architecture (A)</td>
<td>Functional analysis</td>
<td>Functional architecture</td>
<td>Software architectural design</td>
<td>Software architecture description</td>
</tr>
<tr>
<td>Design (D)</td>
<td>Synthesis</td>
<td>Physical architecture</td>
<td>Software detailed design</td>
<td>Software design description</td>
</tr>
<tr>
<td>Coding (C)</td>
<td>(None)</td>
<td>(None)</td>
<td>Coding and unit testing</td>
<td>Code</td>
</tr>
</tbody>
</table>

Table 1. The Activities of Systems Engineering and Software Engineering and Their Associated Activities and Artifacts
The vertical axis in Figure 1 is labeled “engineering,” to reflect the course of the engineering process described above, and its direction indicates the “progress,” because the design progresses as the engineering process proceeds. Any progress metric may be used, although a common example is the number of requirements being analyzed, approved, designed, and so on. As for software, size metrics such as source lines of code (SLOC) or function points are applicable.

The problem with one-dimensional system development is that it cannot handle such common situations as a change of client’s requirements during development or the detection of a serious fault that necessitates going back and fixing the artifacts of earlier activities before development can proceed. In the context of software-intensive systems, life-cycle models like the once-through strategy that partition the life cycle into modification-free development followed by software maintenance exclude many of the software engineering advances of the past 20 or 30 years, such as the iterative models characteristic of the object-oriented paradigm, let alone the waterfall model, as described in the next section.

3. TWO-DIMENSIONAL SYSTEM DEVELOPMENT

A characteristic of two-dimensional system development is that the design may be modified during development. To see the implications of this, consider the following case study: A new system is being developed for a client. While the design is being produced, the client’s requirements change in such a way that the architecture and design have to be modified before development can proceed. Once the system has been implemented (that is, fabricated, assembled, integrated, and tested), it turns out that the system does not meet certain security constraints, so the design has to be modified; the modified design is then implemented. The resulting product passes its acceptance test and is released for production.

The waterfall model [Royce, 1970] can easily handle such a life cycle. The waterfall model can be viewed as the once-through strategy with feedback loops for handling what Royce called “unforeseen … difficulties” [Royce, 1970]. In other words, if a situation occurs such as those described in the previous paragraph, namely, a change in the client’s requirements or the detection of a serious fault, the development team can follow the feedback loops, make the necessary changes, and then resume development. Feedback loops are also essential in the systems engineering process depicted in [IEEE, 1999: Fig. 4]. The waterfall model is depicted in Figure 2, which explicitly reflects the feedback loops. Other two-dimensional models include the spiral model [Boehm, 1988], the V-Model [V-Model, 1997], and the ZOPH model [Negele et al., 1999]. The problem with Figure 2 is that it does not show which of the feedback loops are followed during the course of the case study, let alone the order in which this is done. Moreover, each of these loops, when followed, produces a modified version of the relevant artifacts of the previous activity. However, these changes in existing artifacts are not reflected in the waterfall model. This problem is rectified in Figure 3, which is based on the evolution tree model [Tomer and Schach, 2000].

The two-dimensional nature of Figure 3 helps to depict the life cycle of the case study. The leftmost column represents the development of the first version of the system design. Starting from scratch (φ), the requirement baseline (RA₀), functional architecture (AA₀), and physical architecture (DA₀) are produced.

![Figure 2. Waterfall development model showing feedback loops.](image)

![Figure 3. Evolution tree model of case study.](image)
But before this version can be coded, the client’s requirements change (the line surrounding CA₀ is dashed, denoting that this activity was never performed). The changed requirements baseline (RA₁) induces changes to the functional architecture (AA₁) and physical architecture (DA₁), and the new design is now implemented by new code (CA₂). Then, the physical architecture is changed again (DA₂) and implemented (CA₂).

“The process of improving a system after production through modification to correct a design deficiency or to make an incremental improvement” is defined in IEEE 1220 [IEEE, 1999: Definition 3.1.32] as reengineering. Although no production is involved, the evolution of artifacts along the horizontal axis reflects closely the definition of reengineering, and therefore the horizontal axis is labeled accordingly. The direction of the reengineering axis indicates the effort of rework invested in the reengineering activity.

The complete set of artifacts comprising the system design is usually referred to as the configuration of that version of the design. For example, just before the client’s requirements changed, the configuration comprises the artifacts RA₀, AA₀, DA₀. The configuration of the first working version is RA₁, AA₁, DA₁, CA₁, and the configuration of the delivered version is RA₁, AA₁, DA₂, CA₂. In other words, in two-dimensional system design, engineering and reengineering intertwine throughout development. Furthermore, there will generally be multiple versions of artifacts. Each version of the system design, however, comprises a unique set of artifacts, the same sort of artifacts that are created in one-dimensional system development. The difference is that two-dimensional system development keeps exact traces of all the versions of system design, along with all their artifacts.

Along the vertical axis, the engineering axis, we can trace the activities during which the various artifacts of a particular version were constructed. Along the horizontal axis, the reengineering axis, we may trace the evolution of a particular artifact (the physical architecture, for example) as new versions of the system are constructed. In the following, we show that this is essential in product lines, where new products are designed on the basis of existing designs.

4. THREE-DIMENSIONAL SYSTEM DEVELOPMENT

A characteristic of classical development is that the development team builds the target product starting from scratch. In contrast, it is generally quicker and cheaper to reuse an existing artifact in a new product than to specify, design, implement, document, and test that artifact from scratch [Henry and Benoit, 1995]. For example, software developers try to utilize existing software artifacts wherever possible. Reuse can be applied at different levels of abstraction. At the concrete component level, electronic cards and chips are frequently reused, either homemade or purchased commercially “off the shelf.” Another example is the reuse of graphics libraries, such as Microsoft Foundation Classes (MFC), when a product is to include a graphical user interface (GUI). Moving to a higher level of abstraction, reuse of design patterns is becoming an increasingly popular way of reducing the effort in designing a software module. This concept, which emerged from the architecture of buildings, is applicable to other engineering disciplines as well. Furthermore, when a design pattern is reused, parts of the physical components that implement the pattern in question may also be reused. In large systems, especially those designed with an eye to scalability and diversity, a reusable architecture is generated, to cater for future similar systems.

There are two types of reuse. In black-box reuse, an artifact is reused unchanged. In glass-box reuse, the artifact is modified before it is utilized in the new product. This modification of an artifact before reuse constitutes reengineering of that artifact.

The artifacts discussed in this paper are only those within the scope of development, that is, no physical components are considered. However, reusing development artifacts (such as architectures, mechanical mechanisms, chemical processes, electronics designs, and software components) is essential for any company seeking new market niches or new clients by utilizing systems previously designed and supplied.

As software artifact reuse became more and more popular, a framework for software product lines was introduced, which defines a software product line as a “set of software-intensive systems ... sharing a common, managed set of features...” [Clements and Northrop, 1999 p. 3]. A substantial feature of this framework is a repository in which reusable artifacts, at all levels, are stored for future reuse at the enterprise level. Because these artifacts serve as building blocks in future products, they are named “core assets.”

In many cases, core assets are not originally created as such, but rather originate in other products in the line. Thus, in addition to the transfer of core assets from the core assets repository to specific products (acquiring a core asset), specific artifacts may be transferred to the repository (mining an existing asset) in order to become core assets.

We can view the life cycle of the system design over the entire product line as shown in Figure 4. The two horizontal axes are associated with engineering and
reengineering, as before. The vertical axis is associated with reuse. Thus, project-level system development is two-dimensional, whereas enterprise-level system development is three-dimensional.

The life cycle of the core assets is depicted on the lowest (earliest) plane. Above it are displayed the life cycles of three products that were constructed by reusing core assets. The position of a life cycle (horizontal plane) with respect to the vertical axis is determined by the time at which the respective product was started. The two dashed arrows represent the two basic operations associated with core assets in software product lines, namely, copying a core asset from the core assets repository to a specific product (reusing or acquiring a core asset), and copying a specific artifact to the repository (mining an existing asset) in order to become a core asset [Schach and Tomer, 2000].

Reengineering is the distinctive characteristic of two-dimensional system development (as opposed to one-dimensional system development). In the same way, reuse is what distinguishes three-dimensional system development from two-dimensional system development. It could be argued that three-dimensional system development should be reserved for those products where reuse is utilized and that simpler two-dimensional system development is adequate for most systems. There are two reasons why we reject this argument. One reason is that reuse is becoming widespread. At the beginning of this section, we mentioned MFC and design patterns, both of which, by all accounts, are being utilized relatively widely. But the major reason why we believe that system development is now three-dimensional is that, in our opinion, a broader form of reuse is being employed in virtually all modern system development. For example, clients try to use their domain knowledge to state reasonable requirements, based on application-domain facts and rules. Analogously, system architects utilize their architectural knowledge when producing architectures, designers utilize widely known design solutions in their designs, and programmers usually make explicit use of software libraries. In short, the days of “development from scratch” are over; reuse is now the rule, rather than the exception. In view of the universal nature of reuse, three-dimensional system development is now a requirement rather than an option.

5. THE THREE BASIC OPERATIONS OF SYSTEM DEVELOPMENT

There are three dimensions to system development, namely, engineering, reengineering, and reuse. Analogously, there are three basic operations of system development, namely, generation, modification, and replication. Let A be an artifact and B be the same artifact after modification, denoted by $B = \text{modify} (A)$. Now let another artifact C be an exact replica of A, denoted by $C = \text{copy} (A)$. Finally, let $D = \text{generate} (A)$ denote the creation of D from artifact A.

Engineering, in the classical sense, is implemented via the generate operator. The generation may be manual or automated. For example, we may generate code from a design either using an automatic code generator or by hand.

Similarly, modify is the main operator involved in reengineering. This follows from the definition of reengineering [IEEE, 1999: Definition 3.1.32]. Other sources ([Chikofsky and Cross, 1990] for example) tend to define reengineering as reverse engineering followed by forward engineering. However, we will show later that this broader definition is described in our model as a compound operator, comprising simpler operators.

In terms of our three-dimensional model depicted in Figure 4, the generate, copy, and modify operators may be described as “moves” of artifacts along the axes; generate takes place along the engineering axis, copy refers to moves along the reuse axis between horizontal planes, and modify refers to moves along the reengineering axis within the same plane.

Figure 4 depicts a three-dimensional model for the evolution of the artifacts of system design in a product line. In fact, any artifact produced during the development of any product can also be represented in the three-dimensional space defined by the above three axes. In other words, each artifact may be characterized by three coordinates, as follows:
• The location of an artifact along the engineering axis is determined by its type. For example, as shown in Figure 3, architecture artifacts are located at an earlier level than design artifacts. The scale and resolution of the engineering axis may vary according to the specific life-cycle model being used. For example, artifacts may be characterized as inception, elaboration, construction, or transition artifacts, thereby supporting the Unified Software Development Process (USDP) life cycle [Jacobson, Booch, and Rumbaugh, 1999], now the Unified Process (UP). They can also be characterized as use-cases, class diagrams, sequence diagrams, or code modules, if UML is utilized [Rumbaugh, Jacobson, and Booch, 1999]. The only constraint is that the types be ordered, in order to determine unambiguously which types are derived from other types in the course of development.

• The location of an artifact along the reengineering axis is determined by its version number. Artifact versions are usually numbered decimally, for example, version 5.02. Version numbering can always be used to determine the time ordering of each version of a specific artifact. Because multiple versions of the same module may be needed (for example, for implementation in different languages or to run under different operating systems), we consider each version of an artifact to be an artifact on its own.

• The location of an artifact along the reuse axis determines the specific product of which the artifact forms part. More specifically, we may assume that every product has its own product library, containing the set of artifacts comprising the system design configuration of that product. In addition, we also assume that there is one special library implementing the core asset repository (represented by the lowest plane of Fig. 4). The direction of the reuse axis indicates the diversity of the product line: The more planes displayed, the more specific products have been developed. Whereas along the other two axes the artifacts are ordered according to the time of their creation, if we were to utilize the exact time, along the reuse axis, at which a specific artifact moved from one plane to another, the product trees of Figure 4 would generally intertwine. In practice, therefore, the position relative to the reuse axis of a specific artifact within a specific product serves more as an illustration rather than as an exact mathematical notation.

Accordingly, each artifact A may be denoted by a triple A = (T, V, P), whose components correspond to the type, version, and product characterizing that artifact. For example, A = (“functional architecture,” 3.12, “RS for Navy”) describes the artifact holding version 3.12 of the functional architecture of a navy version of radar system RS—one of a family of similar radar systems for various armed forces.

We return now to the three basic operators mentioned above. The main reason they are considered basic is that each of them affects exactly one axis. In more detail:

- \((t', v, p) = generate (t, v, p)\) corresponds to a development step within a certain product p, resulting in a new artifact with a “later” type \(t'\), but with a version number \(v\) identical to that of its predecessor.
- \((t, v', p) = modify (t, v, p)\) corresponds to a modification step, resulting in a newer version of the same type of artifact in the same product.
- \((t, v, p') = copy (t, v, p)\) corresponds to a duplication step, resulting in an exact replica of the artifact but within a different product library.

Inverse operators, associated with moves along the same axes, but in the opposite direction, are equally significant. Specifically, reverse engineering, that is, moving to a representation at a higher level of abstraction (for example, creating a design model from the source code) corresponds to \(generate^{-1}\), affecting only the type component. Reverting to a previous version, perhaps by means of configuration management ("rollback"), corresponds to \(modify^{-1}\). Finally, mining corresponds to \(copy^{-1}\).

Combinations of these three basic operators are also intrinsic to three-dimensional system development. For instance, let us consider an example of a software product line. Let \(C\) denote a core asset, and let \(P\) denote an artifact of a product.

(a) \(P = modify (copy (C))\). This is the case where a core asset is copied for reuse in the product and is then modified to produce a specific artifact. It is a classical scenario of reusing a core asset as a baseline for a specific artifact. A typical example is where \(C\) is a design pattern and \(P\) is a specific design based on the pattern. It is likely that \(copy (C)\) remains an asset of the specific product, to document the design.

(b) \(P = copy (modify (C))\). This is the case where a core asset is modified before being acquired by the product. A typical example is where a specific design pattern is developed by the components
An electronic card contains a processor \( P \) running program code \( C \). In order to improve the performance for a new product, the engineers decide to implement part of the functionality assigned to the processor by a designated hardware chip serving as the processor’s “slave” \( S \). The overall functionality of the card is unchanged. The engineers apply the following steps:

- Using a CAD tool, generate a design of \( S \) based on a subset of requirements \( R \), previously allocated to the processor.
- Load \( P \) into the CAD tool, together with the newly generated chip \( S \).
- Generate a new physical design (card layout) for the combined components.
- Modify the code \( C \) to support the use of the new slave.
- Generate a new version of the card by compiling the code to produce an object module and loading it onto \( P \).

The entire process may be described as \( \text{generate} \ (\text{modify} \ (\text{generate}^{-1} (Q))) \).

We now consider the implications of our three-dimensional model for system development.

6. DISCUSSION

In the above, we have shown that system development is far from being a once-through process. After a system has been delivered to its clients, it may be recalled for alterations and improvements. In addition, there are at least three other cases where the system design evolves even before the product is produced and delivered. First, whenever there is glass-box reuse, by definition there is reengineering, and thus an evolution of the design. Second, if the client’s requirements change before development is accomplished, then the resulting changes to the artifacts that have already been built also constitute design evolution. Third, if the development team has to correct a fault before the product is delivered (for example, when a field experiment has failed), then this, too, is a form of design evolution. This implies serious consideration, definition, and control of the repeated changes occurring throughout the development of a system, causing the design to evolve frequently.

The model depicted in Figure 4 reflects all three aspects of system development. The three axes are (pure) engineering, (pure) reuse, and reengineering. Design evolution, which takes place before the product is ready for production and delivery, is reflected by activity in the reengineering direction within the relevant engineering-reengineering plane. White-box reuse is reflected by activity in the reengineering direction within the relevant reuse-reengineering plane. Finally, as previously mentioned, the three basic operators, namely, generate, copy, and modify, together with their respective inverses, encapsulate the activities of system...
development and record the evolution imposed on the system design.

Furthermore, by decomposing each artifact in terms of its three basic characteristics we may follow the detailed history of changes made to that artifact. In the context of product lines, new products do not always evolve from modifications to current products. For example, suppose that a former client requires a modified version, MV (say), of a product that has been in use for long time, whereas the developer has recently delivered upgraded products to newer clients. An appropriate combination of original artifacts together with newer ones might be the best way to build modified product MV. This could be effectively achieved if the change history of each artifact is traced in detail by means of the model we have presented in this paper.

7. CONCLUSIONS AND FURTHER WORK

We have presented a model that is both comprehensive and inclusive. Besides being in line with development processes defined in widely used systems and software engineering standards, it can be used in conjunction with any life-cycle model and any set of artifacts. The key aspect of our model is that it is evolution oriented, that is, it focuses on the importance and significance of the repeated changes made throughout the development stage, instead of relegating the treatment of such changes to a secondary or even tertiary role. As described in the previous section, our model incorporates the system design evolution that takes places after the product has been produced and delivered to its client, as well as three different types of system design evolution that take place before the product is produced.

We have identified the three basic building blocks of system development. The three mathematical operators, copy, modify, and generate, together with their inverses, can be used to describe the various systems engineering activities. In fact, we have shown by means of an example that these operators constitute a simple yet powerful language, which may lead to a sound semantics of system development activities and system design evolution throughout the entire system life cycle. We intend to investigate the possibility of using these semantics for defining a methodology as a whole, as well as the capabilities of computer-aided design tools and configuration management tools. This line of research might lead to a formal framework for system development, which focuses on the evolution of the system design.

Finally, we have shown how our model can be used to trace back the change history of each artifact. This is achieved by identifying the source of each artifact and following the changes to that artifact through the system development activities to which that artifact was subjected. This information can then be used for the development of new products. It is our intention to investigate more efficient ways of storing this information, perhaps within the artifact itself or within the product library.

ACKNOWLEDGMENTS

The work of Stephen R. Schach was supported by the National Science Foundation under Grant Number CCR–0097056.

REFERENCES


Amir Tomer is the Director of Systems and Software Engineering Processes at RAFAEL Ltd., Israel, with whom he has been since 1982, holding a variety of systems and software engineering positions, both technical and managerial. His B.Sc. and M.Sc. degrees in computer science are from the Technion, Israel, and his Ph.D. in computing from Imperial College, London, UK. Amir also teaches software engineering at the Technion and other colleges. E-mail: tomera@rafael.co.il.

Steve Schach is an Associate Professor in the Department of Electrical Engineering and Computer Science at Vanderbilt University, Nashville, Tennessee. He is the author of over 100 refereed research papers. Steve has written eight software engineering textbooks, including Object-Oriented and Classical Software Engineering, 5th edition (McGraw-Hill, New York, 2002). He consults internationally on software engineering topics. Steve’s research interests are in software maintenance and systems evolution. He obtained his Ph.D. from the University of Cape Town.