Software Component Composition: A Subdomain-based Testing-theory Foundation

Dick Hamlet

Portland State University
Portland, OR, USA
hamlet@cs.pdx.edu

Abstract

Composition of software elements into assemblies (systems) is a fundamental aspect of software development. It is an important strength of formal mathematical specification that the descriptions of elements can be precisely composed into the descriptions of assemblies. Testing, on the other hand, is usually thought to be "non-compositional." Testing provides information about any executable software element, but testing descriptions have not been combined to describe assemblies of elements. The underlying reason for the compositional deficiency of testing is that tests are samples. When two elements are composed, the input samples (test points) for the first lead to an output sample, but it does not match the input test points of the second, following element.

The current interest in software components and component-based software development (CBSD) provides an ideal context for investigating elements and assemblies. In CBSD, the elements (components) are analyzed without knowledge of the system(s) to be later assembled. A fundamental testing theory of component composition must use measured component properties (test results) to predict system properties.

This paper proposes a testing-based theory of software component composition based on subdomains. It shows how to combine subdomain tests of components into testing predictions for arbitrarily complex assemblies formed by sequence, conditional, and iteration constructions. The basic construction of the theory applies to functional behavior, but the theory can also predict system non-functional properties from component subdomain tests. Compared to the alternative of actually building and testing a system, the theoretical predictions are computationally more efficient. The theory can also be described as an exercise in modeling. Components are replaced by abstractions derived from testing them, and these models are manipulated to model system behavior.

Keywords: Software components, system assembly, composition of properties, foundational testing theory, component-based software development (CBSD).

1 Components for Software Development

In many engineering disciplines, the idea of aggregating standardized components to create a complex system has allowed the creation of better systems more easily. Component descriptions are catalogued so that a system designer can design a system "on paper." Adequate catalogue descriptions of components are the basis for computer-aided design (CAD). CAD tools help the system designer to predict properties that hypothetical systems would exhibit if built from those components.

The component approach has promise for dealing with the difficulty of design and uneven quality of software systems. Divide and conquer is the only known way to attack the overwhelming complexity of software; software components should be easier to design and analyze than complete systems. Unfortunately, the analogy between electrical/mechanical and software components breaks down when their behavior is considered in detail. The traditional mechanical component has continuous properties that can be described by a handful of measurements, and statistical quality control gives a high probability that any given unit will adhere to the general description. Software, on the other hand, is notoriously difficult to specify, and sampling (testing) does not probe its properties very well, partly because they are discontinuous and partly because too many tests are required.

The central dilemma of software design using components is that component developers cannot know how their components will be used and so cannot assume any particular environment for component testing. Yet a component's behavior depends critically on its environment. A customer for components (the system designer) does know the environment, but the information comes too late. If components must be assessed at system-design time, most of the benefit of component-based development is lost. In practice, of course, a component may be intended for applications whose environment is at least partly known.

1.1 Ideal Component-based Software Development (CBSD)

If the design of software is to benefit from using catalogued components as have other engineering disciplines, there must be a strict separation between component development *per se*, and component use in system development. A component catalogue is the document that effects this separation. It records the work of component assessment in such a way that system designers (and their tools) can perform system-synthesis calculations entirely without access to the components themselves. It is conventional to refer to catalogue entries as "component specifications," although that terminology is not used here because in Section 6.2 it is seen to be something of a misnomer. The point is that for CBSD to work, the catalogue must be precise and accurate: it must tell the system designer everything he/she needs to know about a component and must not be misleading. If the catalogue is inaccurate, the system designer may have to repeat or extend component analysis to get useful system predictions.

It is characteristic of an acceptable component technology that the component catalogue is trusted. People do make mistakes, but it is quite unthinkable that catalogue entries are purposely falsified. When there is a lack of information, trust is impossible. To give a real example, so-called "process metrics" are not acceptable in a catalogue. To say "this component was developed by an SEI level-5 organization," is quite unlike saying, "its failure rate is less than 0.00001/hr;" only the latter would be tolerated in electrical or mechanical engineering. The proper role for subjective measures is to engender belief in precise ones. One may find it easier to believe that an SEI level-5 organization could construct a quality component and could accurately measure its failure rate. It is natural that component developers bear the burden of describing and measuring properties of their products. They compete on the basis of quality and price, and it is in their self-interest to balance these factors and to publish the result so that good work will be rewarded by being selected for system designs.

Given an adequate component catalogue, system design is the creative process of selecting and combining components that should work together to meet the system requirements. Talented designers will do this better than hacks. However, the process is complex and error prone, so it is essential to try out proposed system designs. When there is a theory of composition based on the descriptions in a catalogue, the trial can be done "on paper." The components are not actually assembled, nor are any real tests carried out. Rather, calculations predict what the properties of the system will be. Substituting one component for another in a trial system requires no more than repeating the synthesis calculation with a different catalogue entry;

the predicted system properties for the alternatives can then be compared. It is important that predictive calculations be efficient—much faster than executing an actual system.

The ideal of CBSD is a stringent one, far from being realized in practice. The testing-based theory presented here is able to conform to most of the ideal by utilizing 'components' that are themselves idealized. By imposing extreme restrictions on the component form and on the system architecture, the ideal paradigm can be studied. This form of investigation, in which a simple model is quantitatively examined rather than looking at a more realistic model qualitatively, has a distinguished history that includes Turing's computation model. The goal is not direct application—no one imagines building Turing chips—but rather understanding.

1.2 Component Properties

Historically, "component" in software is a rough synonym for "module" or "unit" or "routine." The word originated as a reference to source code in a programming language, but unfortunately this natural viewpoint leads to inconclusive terminology wars over what definition of 'components' should be used in CBSD. Clemens Szyperski suggests shifting the focus away from code source. He defines a software component as executable, with a black-box interface that allows it to be deployed by those who did not develop it [46]. This paper uses a restricted form of Szyperski's definition, taking a component to be an executable program with pure-functional behavior.

System properties that arise from composing components can be categorized as follows:

Black-box behavior. The so-called 'functional' or 'input-output' behavior of any program is its most important characteristic.

Compositional non-functional behavior. Some component properties such as run time and reliability intuitively combine to yield their system values. These properties are a primary concern of this paper, described in Section 3.

'Emergent' behavior. Other non-functional system properties arise only because components are used together. Security properties such as restricted access to classified information are of this kind. Emergent properties may still be "compositional" as described in Section 5.2.

Without precise component descriptions and a way to use them in predicting system properties, software components may be no bargain. To buy off-the-shelf software with unknown properties is only to trade the difficult task of assessing your own work for the more difficult task of assessing someone else's [48].

1.3 Ideal Components and Systems

Choosing a restricted model of components for a foundational theory is an uneasy compromise between making the model plausible yet simple enough to be tractable. Since the goal here is to carry through a complete quantitative analysis, simplicity is primary. The theory of testing, which began with Goodenough and Gerhart [12], suggests most of the necessary restrictions. First and foremost, testing theory takes programs to have functional semantics. A program is assumed to take an input (conceptually a single value) and produce an output (also a single value). A program is correct if its input-output behavior matches a specification function given *a priori*.

Reliability is an important non-functional property of software. This forces the further restriction to a real-number input domain, since it is difficult to define random sampling for other spaces.

These historical choices almost completely determine constraints on a fundamental component-composition theory. If each component has a real-valued input domain, then to combine them requires their outputs to also be reals.

For the system architecture there are two choices that have historically been explored: (1) Functional composition only, using recursion to handle cyclic computation and Boolean characteristic functions to model conditional computation, or (2) The three 'structured' operations of sequence, conditional, and iteration [3]. The latter is much closer to the mainstream model of computation in imperative languages and it is chosen here.

In summary, in this paper a component has pure-functional semantics and a single real-valued input and output; its non-functional properties are also mappings on the reals; and components form systems using arbitrarily nested sequence, conditional, and loop constructions. Such a system then necessarily obeys the same restrictions, making it a technically a 'component,' a nice closure property.

2 Dilemma of Varying Software Behavior

If software is intrinsically different from products of mechanical engineering, it is because software obeys no natural laws, and therefore lacks the simplifying organization often imposed by nature [19]. Most natural phenomena are continuous and this continuity allows a brief but precise description of a physical system. For example, a mechanical system often has components that can be described as point masses, and Newtonian mechanics can accurately predict the behavior of very complex assemblies from this description alone. Software, in contrast, is usually discontinuous and may have arbitrary human-defined behavior that must be described explicitly in forbidding detail. This fact explains why requirements engineering is so important and so difficult.

The difficulty in calculating system properties from component test measurements can be illustrated by a simple example. Imagine two software components placed in series. The first component C_1 receives the system input, does its calculation and invokes the second component. The second component C_2 does its calculation on input received from C_1 and C_2's output is the system output. Consider the performance property of this composite system. To use the paradigm that has been successful in other engineering disciplines, one wants to measure the run time of each component in isolation and then calculate the system run time. Suppose that each component is capable of 'slow' or 'fast' performance, depending on its input. The system run time will then depend on two things:

1. The distribution of system inputs over the input domain of the first component. For example, if many inputs lead to the 'slow' behavior of C_1 then the system will be slower.

2. The way in which C_1 sends its outputs into the input domain of C_2. For example, if many C_1 outputs happen to fall on 'slow' input points of C_2, the system will be slower.

The usage of a system can be captured by its input profile: a distribution describing how likely it is that each input will occur. Given this distribution, it would be possible to analyze the system above by seeing how many inputs invoke 'slow' or 'fast' behavior in each component and make a detailed, accurate calculation of the composite behavior. But component developers cannot know the profile and cannot know which components will be used together—those are both *system* properties. So how can correct measurements be made at component-development time? This situation is pervasive in software components and systems. It occurs in performance analysis (as in the example) and in reliability estimation. It is no wonder

that engineers from other fields have thrown up their hands at including software in systems-engineering calculations[1].

Software testing theory has a way to divide and conquer problems of disparate and extensive input domains. So-called 'subdomain testing' divides an input domain into a manageable number of subsets (subdomains) with tests selected in each subdomain. There is a substantial literature on subdomain testing[2] beginning with the work of Howden [27, 28]. In software reliability engineering (SRE), subdomains are used in a way that is close to the present purpose. In SRE, functional subdomains are assigned empirical usage probabilities, thus defining a coarse usage profile for a system [39]. Imagining that such a profile will be applied to a component-based system, part of the component-testing dilemma is resolved. The component developer need only supply property values *by the subdomain*. Later, these values can be weighted and combined to get system values, yet the component developer needs no knowledge of the system profile—measurements by the subdomain cover all possibilities.

Testing components in subdomains also resolves the second part of the dilemma, how a system input profile is distorted by one component before it reaches another component. Brute-force tracing of the profile from one component to the next becomes possible because the space is reduced from an intractable number of inputs to relatively few subdomains. In the analysis, each subdomain is like a single 'point,' which makes calculation efficient and allows tractable analysis of loops.

3 Testing-based Theory of System Synthesis

This section presents a quantitative theory that predicts software system properties from component values measured by testing. The theory applies to any software property whose values depend only on the software input and which is mathematically well defined. Performance (run time) and reliability are such properties with numerical (real) values; some security properties can also be incorporated with a bit more difficulty (see Section 5.2). In order to be concrete, the theory will be presented for black-box behavior and for the non-functional property of program run time. Section 3.1 describes component analysis. Section 3.2 gives the rule for composing two components in sequence that is the heart of the theory. Composition rules for conditional and iterative constructions are given in Sections 3.3 and 3.4. Section 3.6 describes calculation of the properties of an arbitrary system.

3.1 Approximating and Measuring Component Properties

A component's run time over input space D is assumed to be a mapping $T : D \rightarrow \mathbb{R}$, where $\mathbb{R}$ is the non-negative reals. Intuitively, $T(x)$ is the run time when the component executes on input x. In testing, T is sampled by executing the component. Suppose that the developer divides a component's input space into a finite number of subdomains $S_1, S_2, ..., S_n$. Sampling on each subdomain and averaging the sample values approximates T as a step function with constant value t_i on subdomain S_i, so that for all $1 \leq i \leq n$, $T(x) \approx t_i,\ x \in S_i$, as indicated in Fig. 1. The vector $<t_1, t_2, ..., t_n>$ = $<t_i>_{i=1}^n$ approximates the run-time function T. For simplicity and clarity, Fig. 1 shows the subdomains as intervals along an axis, which would require the input domain to be ordered.

[1] Nancy Leveson says that when a safety engineer needs to assign a reliability to an embedded software component, it is is usual to take the value as 1.0. She advises that probably 0.0 is more realistic. Neither value is of any use to the system engineer, because the former hides any possible software failure and the latter wipes out all other component contributions.

[2] 'Subdomain testing' has also been called 'partition testing.'

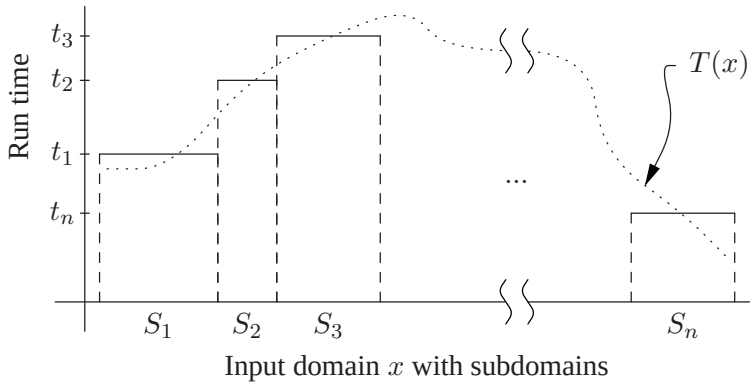


Figure 1: Step-function approximation of a component property

Similarly, assuming the component has an input-output mapping $f : D \rightarrow D$, average values $v_1, v_2, ..., v_n$ of f over each subdomain approximate f with the vector $<v_1, v_2, ..., v_n>$ = $<v_i>_{i=1}^n$. It is also possible to approximate behaviors across subdomains with other non-constant functions. The simplest better approximation is linear, in which the approximating vector contains pairs of (slope, intercept): $f(x) \approx m_i x + b_i,\ x \in S_i$: $<(m_1, b_1), (m_2, b_2), ..., (m_n, b_n)>$, and similarly for $T(x)$. The slopes and intercepts can be obtained by test samples and a least-squares fit over each subdomain. In the sequel, approximating subdomain values by constants will be called the *step-function approximation*, and by linear functions, the *piecewise-linear approximation*. The step-function approximation is just that special case of the piecewise-linear approximation in which all slopes are zero, but it is treated separately because it is intuitively easier to describe. Section 3.5 discusses mixing components with differing approximations.

In any case, the result of a component-developer's analysis effort is a catalogue description consisting of subdomains with functional and run-time values for each subdomain. If the subdomains are well chosen, it may be that this catalogue entry is an accurate description of the actual software behaviors. If not, the accuracy should improve by shrinking subdomain size and aligning functional discontinuities on subdomain boundaries.

3.2 Calculating Properties of a Series System

Suppose that two components B and C are to be composed in a series system U as shown in Figure 2. The information shown in shadowed boxes defines each component by subdomains, input-output values, and run-time values, as measured by testing components B and C, and to be calculated for the composite system U. Figure 2 shows the step-function approximation vectors, as follows: Let the component subdomains be $S_1^B, S_2^B, ..., S_n^B$ and $S_1^C, S_2^C, ..., S_m^C$ respectively (usually $n \neq m$), and let their corresponding output-value vectors be $<v_1^B, v_2^B, ..., v_n^B>$ and $<v_1^C, v_2^C, ..., v_m^C>$. Let their run-time vectors be $<t_1^B, t_2^B, ..., t_n^B>$ and $<t_1^C, t_2^C, ..., t_m^C>$. It is desired to calculate a set of k subdomains for the system U: $S_1^U, S_2^U, ..., S_k^U$, and two corresponding step functions: $<v_1^U, v_2^U, ..., v_k^U>$ for system output and $<t_1^U, t_2^U, ..., t_k^U>$ for system run time.

The calculation derives an 'equivalent component' for the series system U, a 'component' whose description is in the same form as the description of its constituent components. Thus the calculations can be used repeatedly to synthesize the properties of arbitrary systems (see Section 3.6). To calculate the equivalent component means finding a set of subdomains and the input-output and run-time vectors for the system

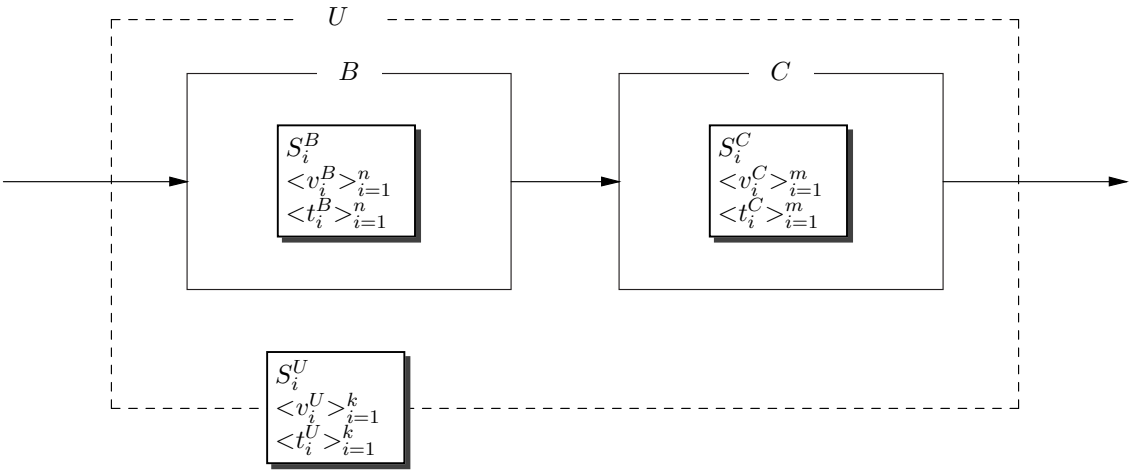


Figure 2: Block diagram of a series system

U.

For the step-function approximation, the system subdomains are those of the initial component B, so $k = n$ and $S_i^U = S_i^B, 1 \leq i \leq n$.

On subdomain S_i^U, B has output v_i^B. Let this fall in the j^{th} subdomain of the following component C. Then the system output value on S_i^U is v_j^C. That is:

$$v_i^U = v_j^C, \text{ where } v_i^B \in S_j^C. \tag{1}$$

The run time of the system on subdomain S_i^U is the run time of B there plus the run time for C on S_j^C:

$$t_i^U = t_i^B + t_j^C, \text{ where } v_i^B \in S_j^C. \tag{2}$$

Using the piecewise-linear approximation is a considerable improvement because it tracks the way in which outputs from one subdomain of component B disperse into distinct subdomains of component C, as the step-function approximation does not. To see the way in which this happens, consider one subdomain interval $S_i^B = [L, R)$ of the first component, in which the functional behavior is described[3] by a line with slope k and intercept q (that is, this line is $\lambda x(kx + q)$). Then the output range is the interval $S' = [kL + q, kR+q)$. This output may fall into several subdomains of the second component. Let one such intersection be with S_j^C and let the linear approximation of the functional value in S_j^C of the second component be $\lambda x(k'x + q')$. Then the equivalent system component has a subdomain that is a reflection back into S_i^B of part of the output interval: $S'' = S' \cap S_j^C$. If this output intersection is the interval $[L', R')$, then the corresponding part of S_i^B is $[(L' - q)/k, (R' - q)/k)$ (if the slope k is 0, the new subdomain is all of S_i^B)[4]. Figure 3 illustrates this subdomain construction. The vertical heavy line is a C subdomain and the horizontal heavy line is a subdomain of the calculated equivalent component, formed by reflecting S'' into the B subdomains. On this new subdomain the composite functional approximation is the composition of the two lines, that is, it has slope kk' and intercept $k'q + q'$. The composite run-time behavior is similarly obtained for the new subdomain, but it is the sum of the linear run-time functions for the components (say

[3]In the interests of readability the presentation avoids subscripts that identify the subdomain.

[4]The derivation is correct only for slope $k \geq 0$. When $k < 0$, the end points of the interval in the second component's domain reverse, and there is a technical difficulty because the right end of the interval is open.

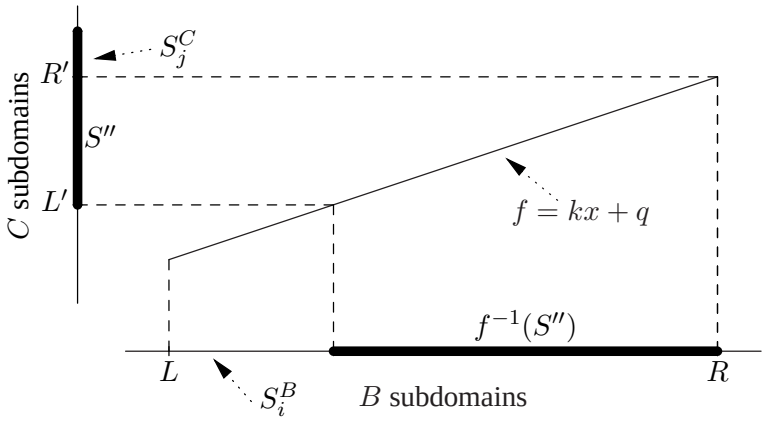


Figure 3: Splitting subdomains in a series synthesis using the stepwise-linear approximation

these are: $\lambda x(hx + r)$ and $\lambda x(h'x + r')$), with the second adjusted to receive an input that is the functional output of the first component. This run-time sum line has slope $h + kh'$ and intercept $r + h'q + r'$.

Repeating this calculation for the intersections between each S_j^C and the output ranges of each S_i^B results in a list of subdomains and linear functions on them for the composite functional and run-time behavior. The piecewise-linear approximation improves the subdomains of the calculated system because whenever a linear output from B crosses a subdomain boundary in C (e.g., at L' in Figure 3), the equivalent component for the series system acquires a new subdomain boundary.

In either approximation, the component data vectors allow calculation of system vectors (from equations (1) and (2) for the step-function case, or from Figure 3 and its discussion above for the piecewise-linear case). System vectors define an equivalent component for the system in exactly the same form as the original component vectors. Hence the calculations can be used repetitively to synthesize arbitrary systems as described in Section 3.6.

The quality of the calculated equivalent component—that is, the degree to which it accurately approximates the actual series system—will of course depend on the accuracy of the approximations for the components. The intuition behind the theory is that as subdomains shrink in size, the approximation should be better and the theoretical predictions should improve. For digital input-output data the space is discrete, so each subdomain contains a finite number of points. Hence the smallest possible subdomains are singletons and in this limit the equivalent-component calculations are exact.

3.3 Conditional System Control Structure

The sequential construction of Section 3.2 can be applied to a conditional:

$$\texttt{if}\ B\ \texttt{then}\ C_T\ \texttt{else}\ C_F\ \texttt{fi}.$$

Let the three components B, C_T, and C_F have subdomains, input-output values, and run-time values[5] using the notation of Section 3.2. Let B have p subdomains, while C_T and C_F have n and m subdomains, respectively.

[5]It is conventional to use the output of B only to determine the branch; whichever of C_T and C_F is selected receives the same input that B received, not B's output.

The conditional test component B partitions the input domain D into:

$$D_T = \{x \in D | B(x)\} \quad \text{and} \quad D_F = \{x \in D | \neg B(x)\}.$$

Input $x \in D$ reaches component C_T iff $x \in D_T$ and similarly input elements of D_F reach C_F. The subdomains of the equivalent component to be computed are therefore:

$$D_T \cap S_i^{C_T}, 1 \leq i \leq n; \;\; D_F \cap S_j^{C_F}, 1 \leq j \leq m. \tag{3}$$

On these subdomains, the input-output behavior of the equivalent component is that of C_T or C_F respectively. The run-time behavior of the equivalent component is that of B in series with C_T or with C_F respectively.

The calculation of an equivalent system component for a condition construction is the same for the step-function and piecewise-linear approximations: the split subdomains carry whichever approximation is being used. D_T and D_F are natural subdomains to use for the conditional-test component B because they exactly capture B's input-output behavior. For input-output behavior it makes no sense to consider B subdomains that cross the *true – false* boundary (that is, $S_k^B \cap D_T \neq \varnothing \wedge S_k^B \cap D_F \neq \varnothing$, for some $1 \leq k \leq p$), which also means that the piecewise-linear approximation for input-output behavior is not meaningful. However, it may be useful for capturing the run-time behavior of B to break D_T and D_F into smaller subdomains or to use a piecewise-linear approximation for B's run time.

A conditional construction with no **else** part is equivalent to taking C_F an identity component with zero run time, which has a perfect piecewise-linear approximation.

Whenever a synthesis construction uses subdomain intersection (that is: for conditionals, in a piecewise-linear series combination, and in iteration because conditionals are used to unroll loops), the count of synthesized subdomains may be as large as the product of the counts for the components. This means that there is a possibility that the subdomain count will grow exponentially in the number of system components. Fortunately, there are some mitigating factors. First, systems are seldom built with more than a handful of components. Second, whenever a series synthesis has a first component with a step-function approximation, it fixes the synthesis count irrespective of the second-component's count. And finally, since the output range of a first component must be contained in the input domain of one that follows, subdomain boundaries tend to line up so that intersections stabilize. The only difficult practical case is piecewise-linear approximation in loop synthesis.

3.4 Iterative System Control Structure

The remaining basic system construct is iteration. Iterative constructions are the bane of program analysis, because in general their behavior cannot be algorithmically obtained in closed form. For this theory things are better than usual. Since there are only a finite number of subdomains, the approximation to loop behavior can be calculated deterministically.

The step-function approximation is easiest to analyze. Begin by unrolling the loop

$$\texttt{while}\ B\ \texttt{do}\ C\ \texttt{od} \quad \text{to} \quad \texttt{if}\ B\ \texttt{then}\ C\ \texttt{fi};\ \texttt{while}\ B\ \texttt{do}\ C\ \texttt{od}.$$

The trailing loop after the unrolled conditional is called the *residual loop*. On any subdomain S_i^C where $B(v_i^C)$ is *false*, the residual loop makes no contribution; these 'false' subdomains can be eliminated from consideration. If there remain C subdomains for which $B(v_i^C)$ is *true*, the loop can be unrolled a second time and further subdomains may disappear because B is false on them. Continuing, at each unrolling at

least one subdomain is eliminated, or none is eliminated. In the latter case, the remaining subdomains are all mapped to one other by C and this situation cannot change, so the approximation to the iterated behavior does not terminate. But unless this occurs, the residual loop will entirely disappear in at most n unrollings, where n is the number of subdomains of C. Thus the equivalent component for an iterative construction is algorithmically determined in the step-function approximation.

For the piecewise-linear approximation, the series composition of the successive unrolled loops can introduce new subdomains as described in Section 3.2, Figure 3. However, the process is necessarily limited. In the worst case, each of C's n subdomains will be split into n pieces; then no further splitting can occur. The argument for the step-function case then shows that at most n^2 unrollings will either eliminate the residual loop or stabilize on a set of 'true' subdomains that never changes so the approximation loop will not terminate. In practice, the worst case is unlikely; even in contrived examples it is hard to force a need for more than about 20% of n unrollings.

If the C subdomains are not fine enough to capture the functional behavior of the loop body well, two difficulties may arise: First, it may falsely appear that C maps out of D_T for some subdomain(s) in equation (3), so that the loop calculation terminates when actually the loop does not. The equivalent component will then be erroneous on those subdomains. Second, the result of executing C may falsely appear to always fall in D_T for some subdomain(s) so that the calculated equivalent component is undefined in those subdomains even though the loop actually does terminate there. The payment for algorithmic loop analysis is that the equivalent-component calculation only approximates the behavior of the iteration construct.

3.5 Combining Different Component Approximations

Although the step-function approximation to component behavior is in general less accurate than the piecewise-linear approximation, steps are sometimes preferred, as in Boolean-valued discontinuous conditional components. Similarly, if there is to be any hope of handling non-numeric functional values, it does not lie with linear approximation. (Section 6.5 provides discussion of a non-numeric case.) Furthermore, the input-output (functional) behavior of a component and its non-functional behavior may not be best approximated in the same way. Run time, for example, always has a meaningful piecewise-linear approximation; reliability (see Section 5.1) does not. Thus there are a number of interesting cases in which components to be combined might have differing approximation measurements. The synthesis theory of the previous sections can be easily adjusted to cover mixed cases.

Input-output functional synthesis makes no use of a non-functional property like run time, so they need not be approximated in the same way. Although run time synthesis does require an input-output approximation, nothing says what form that must take. Hence mixed cases like step-function input-output approximation and piecewise-linear run time approximation require no changes to the synthesis algorithms.

The iteration construction of Section 3.4 is presented in terms of conditionals and sequences, so it requires no modification for a mixed case.

The conditional construction of Section 3.3, once a set of intersection subdomains has been obtained, only reproduces the behavior of the components in its *true* and *false* branches, which may thus be differently approximated.

Only the series construction of Section 3.2 remains. The algorithm for piecewise-linear approximation handles both possibilities, which may therefore be mixed. However, if one of two components in series has a step-function approximation, it does compromise the resulting approximation for the combination. When it is the first component in series that is a step function, there is no subdomain splitting. No matter which of the two has a step-function approximation, the result will be a step function.

3.6 Synthesizing a Component-based System

At the top level of a 'main' imperative program, any system can be built up inductively using the three elementary structured-programming constructions of sequence, conditional, and iteration. The standard software analysis/synthesis paradigm is to:

- Obtain a general rule for each elementary construction in isolation, then
- Perform system calculations piece by piece, using each construction for a given system.

In this way, the largest system is no more difficult to handle than the simplest—it just takes more applications of the three elementary-construction rules.

The rules for constructing an equivalent system 'component' for each of the constructs are given in Sections 3.2 (sequence), 3.3 (conditional), and 3.4 (iteration). To synthesize an arbitrary system, these rules are applied repeatedly. Each time a part of the system is synthesized, it is replaced by a calculated equivalent component, which then enters into subsequent synthesis[6].

It is convenient to describe an arbitrary system structure in reverse Polish notation, using the operators **S** , **C** , and **L** :

Construct	Polish
$X; Y$	XY **S**
`if` Z `then` X `else` Y `fi`	ZXY **C**
`while` Z `do` X `od`	ZX **L**

For example, Fig. 4 shows the Polish representation for an illustrative flowchart and its reduction to an equivalent component (E_4). In the figure, components are named by integers. The final component (E_4 in Fig. 4) has the calculated behavior of the complete system as its functional- and run-time vectors.

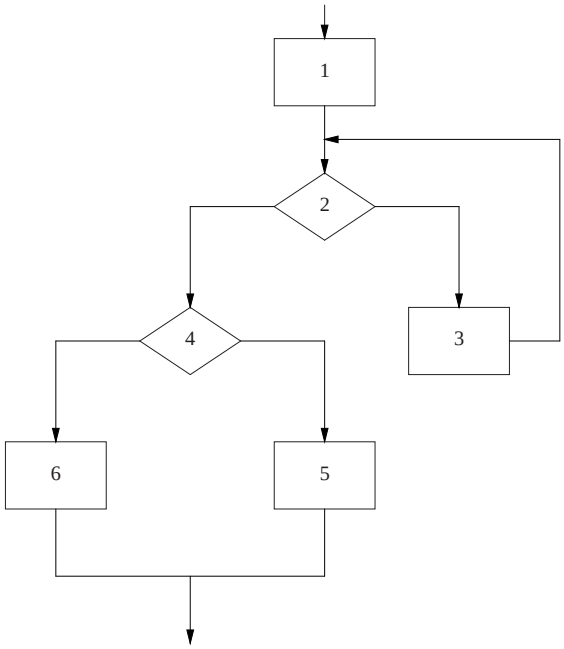
4 Performance of Analysis and Synthesis Algorithms

The usual way to learn about a system's behavior from testing is to build and execute it. The theory of Section 3 provides an alternative: The equivalent component for a system can be calculated and studied. That equivalent component is only an approximation to the actual system behavior, but producing and studying it can be much more efficient than system assembly and testing.

Component analysis itself, the data that goes into system synthesis calculations in Section 3, necessarily uses brute-force sampling. Conventional testing is the only way to obtain component data, and no savings can be expected there. However, it is the whole point of CBSD that the cost of components analysis is borne up-front by their developers, not considered a part of system design. Component measurements are made once and used by all subsequent system designers.

It is in system synthesis that the theoretical calculations of Section 3 can be made much more rapidly than a real system can be executed. Actual run time of a system will depend on execution parameters: the actual run times of components, the number of loop iterations that occur, and the relative frequency with which conditionals take each branch. The theoretical calculations require a time that is independent of these execution parameters, but does depend on the number of system components and the number of subdomains

[6]Although system synthesis was envisaged from the outset, the algorithms given in previous published versions of the theory [22, 23] cannot be used with components that result from previous compositions. This was a mistake that came to light only when supporting tools were implemented.



Evaluate	1 2 3 **L S** 4 5 6 **C S**
2 3 **L** $\rightarrow E_1$	1 E_1 **S** 4 5 6 **C S**
1 E_1 **S** $\rightarrow E_2$	E_2 4 5 6 **C S**
4 5 6 **C** $\rightarrow E_3$	E_2 E_3 **S**
E_2 E_3 **S** $\rightarrow E_4$	E_4

Figure 4: System synthesis by computing equivalent components

chosen. Because the variables are different, only a rough comparison is possible. However, it is shown that the theoretical predictions can in general be arbitrarily faster than actual execution, because they depend only on subdomain counts.

For simplicity, suppose that there are M components in a system, each with roughly the same number of subdomains[7] K. The simplest system structure is all components in series. Let the average run time of components be r. Then to execute the system on one test point requires about rM, or at N points/subdomain, a total time of $NKrM$.

The theoretical calculation does not depend on actual run times. Each step in the synthesis requires a time involving only location of values in subdomain tables. Suppose that whenever a value must be located among K subdomains a time proportional to $K \log K$ is required, for example by binary search, with a proportionality constant (say) α. Then M components in series requires theoretical-synthesis time of about $\alpha M(K \log K)$ in the step-function approximation. The piecewise-linear approximation may split subdomains, but the mechanics of this is that the splitting is self-limiting to a constant factor that could be absorbed into α. The factor MK is common, so the comparison comes down to factors of Nr for actual execution *vs.* $\alpha \log K$ for theoretical synthesis. Either factor can be made to dominate the other since they depend on unrelated variables. Perhaps the best intuitive comparison that can be made is this: If an adequate system test takes a substantial time Nr in each subdomain (say milliseconds) and α is the overhead in a binary search program (microseconds), $\log K$ is on the order of 10^3 for the two factors to be the same. Giving the calculation a hundred-fold advantage makes $\log K$ about 10, or K about 1000. That is, the calculation will be 100 times faster for about 1000 subdomains.

[7]Testing an actual system need not use subdomains, but for comparison the number of system test points is expressed as the product of K and a subdomain sampling frequency.

A conditional construction executes its three components in roughly $2NKr$. The synthesis time for a conditional is essentially that required to intersect two sets of K subdomains, which can be done in $O(K)$ [6, chapter 2]. Thus in the all-series system considered above, replacing three components in series with a conditional gives the calculation an improved edge.

To execute a loop for each iteration requires roughly $2NKr$. The synthesis time for the first unrolling is $O(K \log K)$, and subsequent unrollings require at least one fewer subdomain each, so this is an overestimate if applied to each unrolling. At most K unrollings are required, a total calculation time of $O(K^2 \log K)$. The number of actual iterations may be arbitrarily greater than K, and in the case of an nonterminating loop will be very large just to make a tentative decision that the program is looping. Thus replacing one sequence construct in a system with a loop construct may penalize the theoretical-calculation time by a limited amount if the loop actually iterates fewer than K times, but can penalize the actual-execution time arbitrarily badly otherwise.

In summary, the number of subdomains in the synthesis calculation can be quite large and still give the calculation a large advantage, no matter what form the system takes. If there are long-running components or there is substantial looping, the actual execution time can grow while the theoretical synthesis time remains fixed by the subdomain count. Hence the calculation may enjoy an arbitrarily large advantage. In an example using prototype tools (Section 8.2) the theory is about 25 times faster.

5 Other Non-functional Properties

A quantitative description of component synthesis based on testing has been presented, for functional and non-functional behavior. The theory is more general than the run-time example used to present it, and this section shows how it can be applied to other non-functional properties, both those that are intuitively 'compositional' (e.g., reliability) and some that are 'emergent' (e.g., memory leaks).

5.1 Application to Reliability

Reliability is the basic quality parameter for all engineering artifacts; only software has no generally accepted reliability theory. The reliability application first suggested the subdomain approach [17] and in reference [22] the first version of this theory was presented for reliability alone.

To define software reliability for a program requires an 'operational profile,' a probability distribution over the program's input domain that gives the likelihood of each input occurring in use. Reliability is defined as the probability that if x is selected at random from the operational-profile density, the program meets its specification on input x.

For a component with a collection of subdomains, a reliability step-function can be measured by sampling each subdomain uniformly. Assuming no failures are observed in subdomain S_i for N test samples there, choosing any upper confidence bound c, the reliability is bounded below by t_i [14]:

$$t_i = (1 - c)^{1/N}.$$

Uniform sampling recognizes that nothing distinguishes one point from another within the subdomain. However, for the entire input domain, an operational profile can be expressed as a normalized vector of weights for the S_i, used to form a weighted sum of the t_i, which is a reliability bound for the program over the full input domain.

For components in series, the composite reliability bound for two linked subdomains indexed i and j corresponding to equation (2) in Section 3 is the product:

$$t_i^U = t_i^B t_j^C, \text{ where } v_i^B \in S_j^C. \tag{4}$$

That is, the reliability composition operator is multiplication in place of addition for run time.

The run-time non-functional property was chosen instead of reliability for the exposition of Section 3 only partly because it is more directly intuitive. The deeper reason why reliability synthesis is problematic concerns software reliability theory itself and component independence. In calculating run time for a system the possibility of failures is ignored—it is assumed that the components are computing properly and that measured run-time values are correct. But for reliability, failure is the basic property being investigated. When a component fails, calculations of how it interacts with other components may be incorrect because of that very failure. If a component has a reliability different from 1.0, its functional values must sometimes be wrong no matter what subdomains are taken, and so the equivalent components and equation (4) will be in error, to a degree that is unknown.

Another way to describe the reliability case is to note that equation (4) is a valid combination rule only when the two components are independent. The measured values t_i^B and t_j^C *are* independent, whether they represent run times or reliability estimates. However, the index j is calculated using information (namely, v_i^B) from component B, which links t_j^C to B. Since the mechanism compromising independence is precisely known in this theory, it may be possible to study the correlation. As a beginning, one would expect that for low failure rates of the first component B, it is unlikely that the choice of S_j^C is wrong, hence equation (4) is probably correct—that is, the components appear independent in the theory. This would suggest taking many subdomains S_i^B and sampling them extensively.

Intuitively, reliability is a property that composes subdomain-by-subdomain. If the underlying software reliability theory were more acceptable, the synthesis theory presented here would apply.

5.2 Application to Security and Safety

Testing for the purpose of validating security and safety properties may sometimes be cast as a special case of testing for reliability. The class of failures is restricted to violations of assertions defining security; software security is defined as the probability that a security assertion holds. At the component level this probability can be estimated and a confidence assigned to the estimate when a collection of random tests have been run without violating the security assertion. The corresponding system-level security probability can be calculated as in Section 5.1, with the same caveats about the shortcomings of the underlying theory.

Sometimes security properties only emerge at the system level. This happens when neither of two components has the property, yet it arises from their proper cooperation. Such components may sometimes still be thought to have properties that combine with a system composition operator, but the operator and the properties must be cleverly chosen to result in the emergent property.

Absence of memory leaks is a good example of an emergent yet composable security property. A system is secure in this regard ('leak free') if it never gets memory that it does not later release. If memory is obtained in one component and released in another, the leak-free property can emerge only at the system level. The theory of Section 3 can be applied to a non-functional property that is the set of memory addresses allocated/deallocated by a component. This property *is* compositional, and from it the leak-free property can be determined.

Formally, let M be the set of all memory addresses. Take the non-functional values t_i to be subsets of M allocated and deallocated by a component or a system. That is, $t_i \in 2^M \times 2^M$, where $t_i = (a_i, d_i)$ are

the sets allocated (a_i) and deallocated (d_i). These are net values, internal cancellation having been taken into account: $a_i \cap d_i = \varnothing$. A composition operator[8] "$\oplus$" for components B and C forming a series system U, corresponding to equation (2) for run time is:

$$\begin{aligned} t_i^U &= (a_i^U, d_i^U) \\ &= t_i^B \oplus t_j^C, \text{ where } v_i^B \in S_j^C \\ &= (a_i^B, d_i^B) \oplus (a_j^C, d_j^C) \\ &= ((a_i^B \setminus d_i^C) \cup a_j^C, d_i^B \cup (d_j^C \setminus a_i^B)) \end{aligned} \tag{5}$$

("$\setminus$" is the set difference operator). The rule captures the intuition that net memory allocated by a series composition is that allocated by B and not subsequently deallocated by C combined with that allocated by C, and similarly for net deallocation. The system is leak free iff $\bigcup_i a_i^U \subseteq \bigcup_i d_i^U$. Thus an emergent property can be predicted using the compositional theory.

The archetype emergent security property of maintaining information at different levels of classification can be similarly predicted by keeping access lists for each subdomain in each component.

In security applications, approximations from testing may seem inappropriate. After all, the reason to separate out security properties is their importance and the ability to deal with them using formal mathematical methods. However, testing can quickly catch gross mistakes that occur early in system design. Even if almost every system input violates security it can be difficult to discover this by formal methods.

A nice compromise between sampling (testing) and proof appears in recent work by Jackson on the Alloy system [29] and Boyapati et al. on Korat [5]. They explore an infinite logic-based space of properties by exhaustively testing only an initial finite segment of the space.

6 Theoretical Discussion

This section offers some insights about testing components and about the use of subdomains, based on the theory of Section 3.

6.1 Combining Tested Components with Correct Ones

Mathematical functional analysis of programs in principle solves the problem of calculating any system property T from component properties. Exact system predictions require information equivalent to a correctness proof of each component [38] as well as analysis of its non-functional mapping T. In a system of many components, a few may have been analyzed mathematically; most components will be measured and analyzed using testing methods. Fortunately, it is possible to combine the two kinds of analysis in the step-function-approximation theory.

Let B and C be two components whose properties are defined by testing measurements. Suppose that B is followed by a sequence of proved-correct, analytically described components, which without loss of generality take to be one correct component V. V is in turn followed by C. That is, the component sequence is: $B; V; C$. Because V has been mathematically analyzed, its functional- and non-functional behaviors are known: let $V(x)$ be its output on input x and $T(x)$ its run time.

[8] Equation (5) captures the simplest composition rule—a given address may be allocated or deallocated even it is already in that state. The composition formula that makes it an error to (say) deallocate already-free memory is more complicated. A more accurate operator would make it possible to also determine the emergent property of 'safe memory release' that a system never releases memory that is already free.

To calculate the functional behavior of this series system, in equation (1) of Section 3.2 instead of finding j by where v_i^B falls, use $V(v_i^B)$:

$$v_i^U = v_j^C, \text{ where } V(v_i^B) \in S_j^C, \tag{6}$$

For the system run time, replace equation (2) with:

$$t_i^U = t_i^B + T(v_i^B) + t_j^C, \text{ where } V(v_i^B) \in S_j^C. \tag{7}$$

(Similarly change equation (4) for reliability, but there is no additional factor introduced for $T(v_i^B)$ because reliability for the proved component has value 1.)

Thus component V just extends the functionality of B.

It is disappointing that for the piecewise-linear approximation similar general constructions do not work; the equivalent components do not have linear behavior unless V is linear.

6.2 Test-based, Approximate 'Specifications'

It is usual to say that an engineering component catalogue contains 'specifications' for its entries, descriptions of their properties to be used in selection and in system design. However, 'specification' carries a different connotation for software than for physical components, as a consequence of software's arbitrary nature. A mechanical engineer expects the catalogue description to be all-inclusive, relying on the continuity of physical systems to interpolate across a range of behavior from the published parameters. But for software, any abbreviated description may fail to capture reality. It would be better to qualify the name given to a catalogue description of a software component by calling it an "approximate specification." It has no valid implications that go beyond its explicit limits, and should not be confused with a specification in the sense of an independent description of what software should do, given in some intuitive or mathematical form that includes all cases. For the cases covered by its catalogue 'specification,' the component is by definition correct; it may not be correct for a wider specification in the usual sense if the catalogue description fails to agree with this 'real' specification.

This somewhat peculiar sense of 'approximate specification' occurs in recent work by Ernst on the Daikon system [40] and by Henkel [24]. They induce mathematical 'specifications' from a collection of test data, Ernst in the form of pre- and post-conditions for subroutines, Henkel in the form of algebraic equations for abstract data types. (Henkel calls these "probed specifications.") These 'specifications' have the same character as the catalogue descriptions in this paper, but they are in an entirely different form and obtained by quite different means. Here, testing is the theory behind the measurements; for the others the underlying theory is logical or algebraic and the testing only a device to probe it. It would be of great interest to compare the three forms, but unfortunately their restrictions are largely mutually exclusive. If a program is given only test points on which its input-output behavior is linear, Daikon generates the equation, which matches the catalogue description here on a single subdomain containing those points. But for most simple input-output behaviors that the theory of this paper would capture approximately, Daikon generates only empty pre- and post-conditions. In a personal communication, Henkel indicated that he believes his system would give similar vacuous results on simple examples. Neither mathematical 'specification' system has been applied to non-functional properties.

6.3 Choosing the Right Subdomains

The accuracy of system calculations made from components' catalogue data as described in Section 3 depends heavily on the subdomain breakdown chosen by the component designers. The insight that led testing

theorists to look at subdomains in the first place [28] was that a subdomain should group together inputs that are in some sense 'the same.' One collection of subdomains has the most promise: the path subdomains. For the run-time property the path subdomains are in principle perfect: within one path domain, a constant run time is fixed by the instruction sequence executed[9].

In practical system testing, the most used subdomain breakdown is into so-called functional subdomains derived from the program specification. Each such subdomain comprises those inputs for which the system is specified to perform one intuitive action in the application domain. Functional subdomains are the only sensible basis for so-called black-box testing, but there is no justification for using them to test components in this theory. They describe what a program *should* do, not what in fact it does do.

Unfortunately, true "same" subdomains for input-output behavior have never been investigated. In particular, neither path subdomains nor specification-based functional subdomains seem to bear any relation to subdomains on which functional values are approximately constant.

In its most extreme form, failure to predict the functional behavior of a system arises from the output distribution of the first component in a composition. If that first component spreads its outputs relatively evenly across the input subdomains of a following component, then the tests done by the developer of the latter are valid, because each such subdomain has been systematically sampled. However, if the first component in the composition produces an output profile with a 'spike' in any following subdomain, then the developer's testing of the second component in that subdomain is called into question[10]. As an extreme example, if a component computing the constant function with value K (its output is K for any input) is first in a composition, then the subdomain S_K of the following component in which K falls has a spike at K. Uniform sampling of S_K by the developer of the second component may then be wildly inaccurate unless the second-component subdomain behavior is really constant. As a special case, suppose the second component checks for input K and runs very quickly there, while over most of the rest of S_K it is slow. Then the component developer's average run-time measurement for S_K will be 'slow,' while in this particular system it should have been 'fast.'

6.4 Implications for Testing Practice

Almost every practical testing method falls under the heading of 'subdomain testing,' yet the efficacy of testing by subdomains has proved difficult to validate. Theoretical comparisons [7, 15, 4] between subdomain testing and random testing (disregarding subdomain boundaries) have not shown a conclusive advantage either way in detecting failures. The theory of Section 3 suggests an explanation as follows: Subdomain testing is most often used at the unit level, which is analogous to component-development time in CBSD. Furthermore, only a few unit-test subdomains are selected in a haphazard way and each is sparsely sampled. If subdomain definition and testing as practiced were used on components to make system-level predictions, say of reliability, the results would be very inaccurate. Failure to take care with subdomain boundaries and the paucity of subdomains and samples mean that as a precise unit description, common-practice unit testing is almost worthless. If anything is to be predicted accurately at system level, quite a different order of effort and care would be required.

The eXtreme Programming (XP) methodology [1] presents an even more striking example. In XP, testing from customer scenarios, perhaps with only a few use cases, serves as the only specification from which an incremental design is made. Such a use-case 'specification' is really just a finite description of the

[9] Mason [33] notes that care must be taken with potential run-time errors for this to be correct in principle. In practice there are many factors that can make the same sequence of instructions take differing times to execute, e.g., cache behavior.

[10] A similar difficulty occurs in random system testing when a uniform profile is used because a user profile is not available, as discussed in reference [18].

kind described in Section 6.2, and thus should not be trusted as an oracle unless it includes far more cases than is common.

Testing using subdomains and inducing an approximation to component and system behaviors seem to be good ideas, but research is needed on what works and why, with the probable result that current practice must be much improved to have significant value. It is implicit in an algorithmic-synthesis theory like this one that unit (i.e., component) testing should carry the main burden of verification. Reference [20] explores the relationship between unit- and system testing, suggesting that system testing could be better done on a synthesized model than by actual construction and testing.

6.5 Functional and Non-functional System Properties

The restrictions of the approximation model presented in Section 3 are necessary for the powerful synthesis algorithms given there. These algorithms are difficult enough in the restricted case; it is impossible to imagine that algorithmic synthesis could be carried out using a 'practical' model such as UML. But does this theoretical limiting case have any hope of practical application? Real components seldom compute numerical functions; and approximations, even very accurate ones, cannot be used to check agreement with exact specifications. There is, however, one situation in which both of these difficulties might be overcome.

Non-functional properties like run time and reliability do have real output domains; and for them *bounds* on behavior are sometimes all that is needed. For example, it may be useful to predict from synthesized approximations that the system run time (response) will likely be less than 1 s. There remains the difficulty of dealing with non-numeric input and non-numeric functional output which enters the run-time synthesis. For example, consider character input/output. It may be that the functional approximation, though in principle meaningless, is good enough to be used in the non-functional calculations. A component returning a character value might be 'approximated' as returning an 'average' of (say) `'m'`, because the run time does not depend strongly on the character value, and subsequent components' run times are similarly insensitive to the character received. Thus the functional approximation may be good enough to obtain accurate predictions of interesting non-functional bounds.

7 Related Work

Most of the research needed to realize the promise of software components is properly concerned with creating, combining, and deploying components themselves. Research on qualitative CBSD draws on the communities of software reuse, software architecture, and even systems and networks. Perhaps the most interesting thread in this extensive research literature is the work based on software architecture originated by Garlan [2] and others. In the architectural abstraction, components interface to each other through connectors, which both enable and constrain the component interactions. In the work of Medvidovic and his students [35], the abstract model is implemented as a framework in which a particular style and its connectors are a practical basis for component combination.

In contrast, this paper focuses on the quantitative compositional aspects of CBSD, how component properties (particularly non-functional ones) synthesize to system properties. Approaches to this property-synthesis aspect of component-based systems are the ones surveyed in this section.

7.1 Proof-based and Analytical Theories

In principle, excellent explanatory theories of component functional composition have been available since the late 1960s, in the work of Floyd [8] and Hoare [25, 26], Guttag [13] and others [10, 11], Mills [38, 9],

and many others. In these theories a component is described mathematically, by a collection of logical assertions, or by an algebra, or functionally. The mathematical descriptions have a syntactic interface part and a semantic part completely describing behavior. Component properties such as run time are described by analytical equations in the input variables. The construction of component-based systems might then be described as follows[11]:

> Components can only be used together if their interfaces match, and this match may include semantic properties. For example, one component may require that a list delivered to it by another be sorted so that binary search can be applied to the list. Once components are properly matched, the functions describing their input-output behavior and their non-functional properties can be mathematically composed to obtain the system properties.

These theories are elegant and they completely solve the problem of synthesizing system behavior from component behavior.

Logic-based theories following Hoare [25] have received the most development. Bertrand Meyer proposed 'design by contract' [37] as the form appropriate to reusable components. A version specifically directed at components-based system design and the possible need to modify the pre- and post-conditions of contracts is being investigated by Reussner and Schmidt [42, 44].

7.2 Testing-based Models

Rosenblum [43] has taken a unique approach to relating component tests to system properties, based on very general axioms describing test properties. His work is in a sense at the opposite pole from that reported here. By using only very weak axioms, he achieves the generality that the theory presented here lacks, but correspondingly the results are much weaker than ours.

Karl Meinke [36] uses testing-based program approximation in a different way to seek a failure point for a program P. He requires a first-order formal specification and searches for a failure point roughly as follows: Given a function f that is a piecewise polynomial approximation to the input-output behavior of P, there is an algorithm to search for a point x such that $f(x)$ is incorrect according to P's specification. Let y be the output of P on input x. Perhaps y is incorrect; if so, Meinke has succeeded. If not, he adds (x, y) to f to form f', a more detailed approximation. The process is iterated until either a real failure is found or the piecewise approximation becomes so accurate that the tester believes there are no real failures. Meinke's procedure produces a set of subdomains (the 'pieces' of the approximation) that can claim to objectively capture a 'functional' breakdown of P's domain, itself an important theoretical accomplishment. The work presented in this paper does not require a formal specification, but the payment is that here a tester must construct subjective subdomains by hand. Meinke uses piecewise approximations of order higher than linear since he is not concerned with composing them.

The bulk of prior work on composing component properties in testing theory uses the reliability property. Markov-chain models are used to describe a system as a collection of transition probabilities for invoking each component, and when invoked the component contributes its reliability value. Littlewood's seminal paper [32] appeared long before "component" was a popular buzzword. Mason and Woit [34] obtained good results from decomposing the UNIX `grep` utility into elements that resemble functional-programming units.

Markov models of systems composed of components hide the operational-profile problem described in Section 2 in their assumptions about transition probabilities. Most models begin with a fixed system

[11] The description is not attributed to any particular author, but paraphrases descriptions that occur in work (for example) of the ADJ group [10].

architecture. The transition probabilities can then be measured from expected-usage data for the system. Krishnamurthy and Mathur [30] do not explicitly use a Markov model, but determine the path probabilities in a system by exercising it with a collection of tests. Singh et al. [45] and Kubal, May, and Hughes [31] use a Bayesian approach, beginning with guesses for the transition probabilities and refining these as system test cases are run. To obtain accurate Markov models, the states of a simple control-flow model must be split to account for data-varying transition probabilities, particularly in loop control. It is difficult to obtain plausible probabilities for the split-state transitions and there is a state explosion.

If a theory is to predict system properties from component properties, the latter must be measured in isolation outside the system. In most models, this is done with a fixed operational profile for each component. Thus component reliabilities are single numbers which are then assumed to be appropriate for use in any system in any position within the system. Some models [30] do a little better, measuring component reliabilities in place for the operational profile given to the system. With enough data this approach can be defended, but in the limit it amounts to simply testing the composite system without any independent component measurements.

7.3 Summary of Related Work

Most work cited in this section falls in two categories:

1. Mathematical, analytical methods are entirely correct in principle but are seldom chosen over testing and measurement.

2. High-level modeling of component-based reliability is meant to apply to real systems, but not to aid understanding of component composition itself.

In contrast, the approach taken here is based on testing measurements and the model is chosen to capture as much detail of the component-based system as possible, for the purpose of explaining and understanding what takes place.

A striking feature of this theory is that it treats a number of non-functional properties uniformly. Within the restricted context given in Section 1.3 it applies to run time, reliability, or any property for which a composition operator can be defined—even emergent properties—in essentially the same way.

The connection between this theory and formalism-based work that uses testing in a subsidiary role has been briefly mentioned at the end of Sections 5.2 and 6.2.

8 Follow-on Work

This paper is limited to the presentation of a subdomain-testing-based theory of component composition of pure-function behaviors. The theory was developed in parallel with implementation of research-prototype tools. Theory and tools have been extended to include component local state and validation experiments have been conducted using the tools.

8.1 Components with Persistent Local State

Persistent state is a necessary feature of useful components that is not captured by the model presented in Section 3. Software reliability engineering handles state by making state variables ‘hidden inputs’ and adding them to the input space for random testing. This treatment is simply wrong: state variables are not independent, because their values are created by the software itself and completely out of testing control.

A correct treatment tests software with state by initializing the state and then subjecting the software to a *sequence* of inputs, which results in program creation of a corresponding state sequence.

To model components with local persistent state, a state set H is considered along with the input domain D, and the output and run-time functions map (input×state) pairs: $f : D \times H \rightarrow D$ and $T : D \times H \rightarrow \mathbb{R}$. The state itself is mapped by a new function $g : D \times H \rightarrow H$. Instead of step functions in one input, the approximations to be measured for a component are step plateaus in two inputs.

To give a flavor of the state-inclusive theory, consider synthesizing a series combination of components B and C to form a system U as in Fig. 1 of Section 3.2. In addition to its input subdomains $S_1^B, S_2^B, ..., S_n^B$, component B has state subdomains $H_1^B, H_2^B, ..., H_{n'}^B$, and similarly for m' state subdomains of C. The series system has the same input subdomains as B, and the system has state subdomains that are cross products of the two component state-subdomain collections, e.g., $H_3^B \times H_2^C$. Then to synthesize U, consider its arbitrary subdomain $X = S_i^B \times H_{i'}^B \times H_{i''}^C$. On subdomain $S_i^B \times H_{i'}^B$, B has output v and run time t from its step-plateau approximations. Let v fall in subdomain $Y = S_j^C \times H_{i''}^C$, that is, in the C subdomain corresponding to B's output, but to the C-state part of arbitrary system state X, namely $H_{i''}^C$. Then the system output and result-state values for X are obtained from C's step-plateau approximations evaluated in Y; the system run time in X is the sum of the B run time t and the C run time in Y.

Similar modifications of the constructions in Sections 3.3 and 3.4 yield a system-synthesis algorithm for arbitrary systems made from components with local state.

Adding state raises an interesting new aspect of component testing: portions of the two-dimensional input–state space may be unreachable in principle. In theories based on logic, unreachable states are those in which invariants are violated; it becomes a proof burden to show that they cannot occur, that is, that invariants are preserved. However, it is usual to test states without regard for their feasibility: the state is initialized systematically to values it 'should' have according to a specification. This systematic sampling can create spurious executions that lead the tester astray. The correct sampling using sequences of inputs, however, may fail to excite some hard-to-reach state that is feasible. It is an unsolvable problem to identify exactly the feasible states.

Details of a persistent-state theory that extends that presented in Section 3, and some experiments with component tests and system synthesis are reported in two recent conferences [20, 21].

8.2 Tools and Experiments

The theory presented in Section 3 and its extension to include local state indicated in Section 8.1 have been implemented in tools that measure the step-plateau approximation (input×state)-output, -run-time, and -result-state vectors for components, using subdomains provided by the component developer. With these component descriptions, other tools can synthesize an approximation to an arbitrary system. The synthesis tool is a CAD tool for system design by calculation.

The tools have been used to conduct experiments in component composition, especially to investigate the shrinking of subdomains around rapidly changing or discontinuous behavior. The experiments reveal the way in which the theory successfully approximates system behavior using component approximations. Subdomain refinement generally produces accurate system predictions. For example, in a typical stateless system of six components requiring all of the synthesis rules, keeping the r-m-s component measurement errors below 1% confines the system prediction error to about 4%. A typical time comparison for this example using a 1.7 Ghz PC is 23 s for actual system execution and .9 s for the synthesis calculation. It is surprising how quickly system behaviors become extremely complex even when their component behaviors are simple. The way in which a discontinuity behaves as the subdomain enclosing it shrinks is also unexpected.

A companion paper [16] describes these tools and experiments in detail.

9 Future Work

Among many additional lines of investigation raised by this work, two stand out.

9.1 Error Analysis

It is to be expected that the error in predictions of system properties obtained from this synthesis theory will decrease as the approximation of component behavior improves. However, no theoretical quantitative relationship has been established between approximation and prediction. A component developer cannot know how accurately each component must be described—that depends on its unknown application. But component errors could be used at system-design time to estimate the error in the predicted behavior. "Safety factors" and practical "rules of thumb" are possible when an error analysis identifies the relevant factors, even if precise error predictions cannot be obtained.

A useful error analysis is likely to arise from the study of rapidly changing component behavior. Where component functional behavior changes slowly across a subdomain, the approximations are good. Sampling can never be sure that a discontinuity or rapid behavior change has not been missed, but by shrinking the subdomains, the region of inaccuracy can be confined.

An error analysis for the reliability application (Section 5.1) is the most important and the most difficult. However, it could be that the very difficulties with composition of reliability values can help with error analysis. A component's failure rate is an estimate of how likely it is to produce an erroneous functional value, so it might be possible to carry a reliability calculation as an error predictor along with the calculation of any other non-functional property. Weyuker and Weiss [47] have devised a reliability theory that includes a measure of the functional error, which may be just what is needed for analysis.

9.2 Concurrent Theory

The theory presented here is a sequential one. Extension to concurrency requires adding a concurrency operator to the composition algebra and synthesizing an equivalent component for processing in parallel. Such an addition would greatly complicate the present theory; however, the reliability version of a concurrent theory could be used to discuss methods that seek to improve reliability through redundancy. A voting scheme based on multi-version programming (MVP) is the only known means of adding redundancy to a system and increasing its reliability over that of its components. The question of independence is critical and can only be discussed within a concurrent theory.

The DOTS (Diversity Off The Shelf) project is a large-scale research effort investigating the foundations of MVP, for example in reference [41].

10 Summary and Conclusions

A fundamental, testing-based theory of component composition has been presented. It approximates functional and non-functional behavior of stateless components using subdomain testing, and uses these approximations to calculate approximate properties of systems built from the components. This theory can also be viewed as a mathematical model of component composition, in which step-function approximations model actual components and systems.

Perhaps the most important insight to emerge from the theory is that accurate subdomain boundaries are critical to accurate test-based descriptions. When rapid behavior changes (including discontinuities) occur inside a subdomain, the predictions involving that subdomain will be poor. Component developers cannot

know where crucial system subdomain boundaries will fall, but they should refine subdomains to capture component behavior as accurately as their resources allow.

The implication for conventional unit testing is a disturbing one: since in practice a very few subdomain tests are often used as a quality measure, the confidence gained should not be trusted.

Acknowledgements

Dave Mason and Denise Woit are responsible for the idea that component properties should be *mappings* later used by system developers to obtain values, not values themselves. This insight was crucial to the development of the theory presented here.

This work was supported by NSF ITR grant CCR-0112654 and by an E.T.S. Walton fellowship from Science Foundation Ireland. Neither institution is in any way responsible for the statements made in this paper.

References

[1] Extreme programming: A gentle introduction. `www.extremeprogramming.org`.

[2] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Soft. Eng. Methodology*, 6:213–249, July 1997.

[3] C. Boehm and G. Jacopini. Flow diagrams, turing machines, and languages with only two formation rules. *Comm. of the ACM*, 9:366–371, 1966.

[4] P. Boland, H. Singh, and B. Cukik. Comparing partition and random testing via majorization and schur functions. *IEEE Trans. on Soft. Eng.*, 29:88–94, 2003.

[5] C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on java predicates. In *Proceedings ISSTA '02*, pages 123–133, Rome, 2002.

[6] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry, 2nd edition*. Springer, 2000.

[7] J. Duran and S. Ntafos. An evaluation of random testing. *IEEE Trans. on Soft. Eng.*, 10:438–444, 1984.

[8] Robert W. Floyd. Assigning meanings to programs. In *Proceedings Symposium Applied Mathematics*, volume 19, pages 19–32. Amer. Math. Soc, 1967.

[9] J. Gannon, D. Hamlet, and H. Mills. Theory of modules. *IEEE Trans. on Soft. Eng.*, 13:820–829, 1987.

[10] J.A. Goguen, J.W. Thatcher, and E.G. Wagner. An initial algebra approach to the specification, correctness, and implementation of abstract data types. In R.T. Yeh, editor, *Current Trends in Programming Methodology*, volume 4, pages 80–149. Prentice Hall, Englewood Cliff, NJ, 1978.

[11] Joseph A. Goguen. *Software Engineering with OBJ – Algebraic specification in action*. Kluwer Academic Publishers, 2000.

[12] John B. Goodenough and Susan L. Gerhart. Toward a theory of test data selection. *IEEE Trans. on Soft. Eng.*, 1:156–173, 1975.

[13] John V. Guttag. Abstract data types and the development of data structures. *Comm. of the ACM*, 20:396–404, 1977.

[14] D. Hamlet. Random testing. In J. Marciniak, editor, *Encyclopedia of Software Engineering*, pages 970–978. Wiley, New York, 1994.

[15] D. Hamlet and R. Taylor. Partition testing does not inspire confidence. *IEEE Trans. on Soft. Eng.*, 16:1402–1411, 1990.

[16] Dick Hamlet. Tools and experiments for a testing-based investigation of component composition. submitted to ACM TOSEM, 2006. Copy at: `www.cs.pdx.edu/~hamlet/TOSEM.pdf`.

[17] Dick Hamlet. Software component dependability, a subdomain-based theory. Technical Report RSTR-96-999-01, Reliable Software Technologies, Sterling, VA, September 1996.

[18] Dick Hamlet. On subdomains: testing, profiles, and components. In *Proceedings ISSTA '00*, pages 71–76, Portland, OR, 2000.

[19] Dick Hamlet. Continuity in software systems. In *Proceedings ISSTA '02*, pages 196–200, Rome, 2002.

[20] Dick Hamlet. Subdomain testing of units and systems with state. In *Proceedings ISSTA 2006*, pages 85–96, Portland, ME, July 2006.

[21] Dick Hamlet. When only random testing will do. In *Proceedings First International Workshop on Random Testing*, Portland, ME, July 2006.

[22] Dick Hamlet, Dave Mason, and Denise Woit. Theory of software reliability based on components. In *Proceedings ICSE '01*, pages 361–370, Toronto, Canada, 2001.

[23] Dick Hamlet, Dave Mason, and Denise Woit. Properties of software systems synthesized from components. In K-K. Lau, editor, *Case Studies in Computer-based Software Engineering*, chapter 6. World Scientific, 2004.

[24] J. Henkel and A. Diwan. Discovering algebraic specifications from java classes. In *Proceedings ECOOP '03*, Darmstad, 2003. The authors recommend `www-plan.cs.colorado.edu/henkel` because the proceedings is garbled.

[25] C. A. R. Hoare. An axiomatic basis for computer programming. *Comm. of the ACM*, 12:576–585, October 1969.

[26] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.

[27] W. Howden. Methodology for the generation of program test data. *IEEE Trans. Computers*, 24:554–559, 1975.

[28] W. E. Howden. Reliability of the path analysis testing strategy. *IEEE Trans. on Soft. Eng.*, 2:208–215, 1976.

[29] Daniel Jackson. Alloy: a lightweight object modeling notation. *ACM Transactions on Soft. Eng. Methodology*, 11:256–290, April 2002.

[30] L. Krishnamurthy and A Mathur. The estimation of system reliability using reliabilities of its components and their interfaces. In *Proc. 8th ISSRE*, pages 146–155, Albuquerque, NM, 1997.

[31] Silke Kubal, John May, and Gordon Hughes. Building a system failure rate estimator by identifying component failure rates. In *Proc. 10th ISSRE*, pages 32–41, Boca Raton, FL, 1999.

[32] B. Littlewood. Software reliability model for modular program structure. *IEEE Trans. on Reliability*, 28(3):241–246, August 1979.

[33] Dave Mason. *Probabilistic Program Analysis for Software Component Reliability*. PhD thesis, University of Waterloo, October 2002.

[34] Dave Mason and Denise Woit. Software system reliability from component reliability. In *Proc. of 1998 Workshop on Software Reliability Engineering (SRE'98)*, Ottawa, Ontario, July 1998.

[35] N. Medvidovic, N. Mehta, and M. Mikic-Rakic. A family of software architecture implementation frameworks. In *Proc. 3rd IFIP working int. conf. on software architectures (WICSA)*, pages 221–235, Montreal, Canada, 2002.

[36] Karl Meinke. Automated black-box testing of functional correctness using function approximation. In *Proceedings ISSTA '04*, Boston, 2004.

[37] Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, 2000.

[38] H. Mills, V. Basili, J. Gannon, and D. Hamlet. *Principles of Computer Programming: A Mathematical Approach*. Allyn and Bacon, 1987.

[39] J. D. Musa. Operational profiles in software-reliability engineering. *IEEE Software*, 10:14–32, 1993.

[40] J.W. Nimmer and M.D. Ernst. Automatic generation of program specifications. In *Proceedings ISSTA '02*, pages 229–239, Rome, 2002.

[41] P. Popov. Reliability assessment of legacy safety-critical systems upgraded with off-the-shelf components. In *Proc. SAFECOMP '02, LNCS 2434, Springer*, pages 139–150, Catania, Italy, 2002.

[42] Ralf H. Reussner, Heinz W. Schmidt, and Iman Poernomo. Reliability prediction for component-based software architectures. *Journal of Systems and Software – Special Issue on Software Architecture - Engineering Quality Attributes*, 66:241–252, 2003.

[43] David Rosenblum. Adequate testing of component-based software. Technical report, Irvine, CA, August 1997.

[44] H. Schmidt. Trustworthy components–compositionality and prediction. *Journal of Systems and Software – Special Issue on Component-based software engineering*, 65:215–225, 2003.

[45] H. Singh, V. Cortellessa, B. Cukic, E. Gunel, and V. Bharadwaj. A bayesian approach to reliability prediction and assessment of component based systems. In *Proceedings 12th International Symposium on Software Reliability Engineering*, pages 12–21, Hong Kong, PRC, November 2001.

[46] Clemens Szyperski. *Component Software*. Addison-Wesley, 2nd edition, 2002.

[47] S. N. Weiss and E. J. Weyuker. An extended domain-based model of software reliability. *IEEE Trans. on Soft. Eng.*, 14(10):1512–1524, 1988.

[48] Elaine Weyuker. Testing component-based software: A cautionary tale. *IEEE Software*, pages 54–59, 1998.