Software Component Composition:
A Subdomain-based Testing Foundation

Dick Hamlet*

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

Abstract

Development of software using off-the-shelf components seems to offer a chance for improving product quality and developer productivity. This paper presents an elementary theory of system synthesis from simple components, describes tools that implement the theory, and presents experiments that reveal problems and issues in component composition.

Subdomain testing is the basis of the theory. Component developers describe their components by measuring approximations to functional and non-functional behavior on a finite collection of subdomains. Systems designers describe an application-system structure by the component connections that form it. From component specifications and a system structure, the theory predicts functional and non-functional behavior of the application. The calculations

*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.
are made by a CAD tool that synthesizes the system properties. The system is not built, nor are any test executions performed. Neither the component sources nor executable code are needed by the systems designer. Furthermore, the CAD tool is much more efficient than it would be to assemble and execute an actual system. The theory interfaces with formal verification so that a mixture of tested and proved components can be utilized in system design.

**Keywords:** Software components, system synthesis, 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 assemble a system in the abstract. The catalogue not only describes what each component does, but equally important gives constraints that allow the system designer to decide if a particular component is ‘good enough’ for the application. Accurate, precise catalogue descriptions of components are the basis for computer-aided design (CAD). CAD tools allow the system designer to work with components ‘on paper’ and 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 construct correctly than complete systems. Unfortunately, the analogy between electrical/mechanical and software components breaks down when their behavior is considered in detail. The traditional engineering
component has properties that can be described by a handful of measurements, and statistical quality control gives a high probability that any given sample will adhere to its specification. Software, on the other hand, is notoriously difficult to specify, and testing does not probe its properties very well.

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 the component behavior depends critically on its environment. A customer for components (the system designer) does know the environment, but if components must be measured and assessed at system-design time, much of the benefit of component-based development is lost.

A theoretical solution to the dilemma is presented in a restricted case of testing-based component specification. The component developer performs measurements and provides a component description in such a way that the component buyer can later factor in usage information without repeating the measurements. The buyer designs a system not by construction and trial, but with a CAD tool to make predictions of how the design will behave.

1.1 An Ideal of 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. The component catalogue is the document that effects this separation. It records the work of component assessment in such a way that system designers (and more important, the tools they use) can perform system-synthesis calculations entirely without ac-
cess to the components themselves. It is conventional to refer to catalogue entries as “component specifications,” although in Section 4.3 this is seen to be something of a misnomer. The point is that for CBSD to work, the catalogue must be both accurate and precise: it must tell the system designer all that she needs to know about the component, and must not mislead her. If the catalogue is imprecise, the system designer may have to repeat or extend component analysis; an inaccurate catalogue entry may cause a sound system design to fail. In any case, the system designer has only the catalogue and must be able to work from it alone.

It is characteristic of an acceptable component technology that the component catalogue is trusted. Of course, people can make mistakes, but it is quite unthinkable that catalogue entries are purposely falsified. When there is a lack of precision, 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 synthesis based on the specifications of 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 with the alternatives can then be compared. These synthesis calculations may be tedious, so computer support is essential. CAD tools should do the work, taking as input a description of the system and the catalogue descriptions of components that comprise it. It is important that CAD tools be efficient—the calculations must be fast compared to 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, but only by utilizing ‘components’ that are themselves highly idealized. By imposing extreme restrictions on the component form and on the system architecture, we are able to study the ideal paradigm. 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 the computation models of Turing and others. The goal is not direct application—no one imagines that we should build Turing chips—but 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 [47].

In this paper we use a restricted form of Szyperski’s definition, taking a component to be an executable program with pure-functional behavior.

Most research in components is concerned with specification, design, reuse, and cataloging of the components themselves. Non-functional properties are acknowledged to be important, but are left for measurement and analysis at the system level. The complementary issue of the way in which component properties (particularly non-functional ones) quantitatively combine to appear as system properties has received less attention, but is equally important.

Two apparently different kinds of system properties arise from composing components. One is a property that is intuitively “compositional,” of which run-time performance is typical. Each component has a well defined run time (part of its specification in the ideal) and the run times of interacting components combine (by addition, but with complications discussed in Section 2) to produce the system run time. Most of the exploration of system synthesis to follow is devoted to this kind of property using the run-time example. Reliability is another important system property that arises directly from component reliabilities; its basic composition operator is multiplication. Functional behavior itself is the fundamental compositional property, which in the simplest case is literally functional composition.

The other kind of system property is usually called “emergent,” arising only because the components are used together. Security properties such as restricted access to classified information are of this kind. Emergent properties may still be “compositional” in the sense that there are system
operators that put together component behaviors to form the emergent property. For example, in Section 7.4 the emergent property of being memory-leak free is synthesized. Components might not be leak-free, but their individual memory-allocation behaviors can be combined to establish whether a system is leak-free. The relevant component behavior and the composition operators themselves are more complicated in the emergent case.\footnote{Some emergent properties, for example, concurrent-system properties such as starvation, fairness, deadlock, etc., are apparently not compositional even in this wider sense.}

Thus in ideal CBSD, composition operators for system properties of interest are provided, relevant analysis or measurement is done in isolation at the component level and the results catalogued as component specifications, then synthesis of system properties merely applies the composition operators to the catalogue descriptions.

Without precise component specifications 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 [50].

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 our goal is to carry through a complete quantitative analysis, simplicity comes first. The theory of testing, which can be said to have begun with Goodenough and Gerhart’s paper [12], suggests most of the restrictions. First and foremost, testing theory has taken programs to have functional semantics. A program is assumed to take an input (conceptually a single value, although this value might encode a vector),
and produce an output (also a single value). In the testing model, program state is by encoding sequences of values as a single input. Thus a program specification is a pure function from the input domain to the output range, and the program is correct if its input-output behavior matches a specification function given a priori. These are precisely the assumptions we make here, the programs being both executable components and systems formed from them.

Reliability is the primary non-functional behavior of interest in testing theory. This introduces the further restriction to a numerical input domain modelled as real numbers, since it is difficult to define random sampling for more realistic spaces.

These historical choices almost completely determine constraints on the simplest component-composition theory. If each component has a real-number input domain, then to combine them requires their outputs to also be reals. It is natural to make the same assumptions for non-functional properties: each is a function of a real-valued input.

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 [4]. 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, a single real-valued input and output value, its non-functional properties are also functions on reals, and components form systems by arbitrarily nested series, conditional, and loop constructions. Such a system then necessarily obeys the same restrictions, making it a technically a ‘component,’ a nice closure property. The prototype tools use double-precision floating point approximations to real values.
The only justification for these choices is that as the simplest ones in which testing theory can be applied to components and systems, they are natural for a beginning theory. From a practical point of view, however, the components and systems of the theory are woefully short of reality. So-called mathematical software does observe them, but very little commercial software is of this kind. The restriction to numerical inputs and stateless behavior are bad enough, but the most stringent restriction of the simple theory is the hidden one imposed by assuming functional semantics. A functional theory applies to only one of the varied architectures of component connection. In practice, complex patterns of interaction are implemented (by so-called ‘connectors’ or ‘connector types’) among components whose behaviors are difficult to describe functionally.

However, it is also possible to give some arguments for the plausibility of the model. The single-input constraint is not in principle important, since any complex input vector or sequence can be coded into a single value, and no theoretical problems arise from complex inputs that do not have similar problems mirrored in a single-input theory. It could be argued that restricting to real-valued inputs and stateless behavior are similarly no loss of generality, because reals can code non-numeric values and input sequences can model persistent state. But these two circumlocutions are in practice very cumbersome and unnatural. In future work they might be relaxed, as described in Section 9.

Ultimately, the assumptions made in a model are justified by what can be learned from it. If the model illuminates issues and problems that occur in more realistic cases, it earns its keep, so to speak, and the lack of direct practical application is partly excused. In this paper, the simplifying assumptions allow the definition of an ideal CBSD paradigm and the implementation of testing tools to study it.
2 Dilemma of Varying Software Behavior

If software is intrinsically different from products of traditional engineering, it is because software obeys no natural laws, and therefore lacks the simplifying organization often imposed by nature [17]. Most natural phenomena are approximately 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 can only 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 properties 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 a system 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 the 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.

Software testing provides 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 beginning with the work of Howden [25, 26]. 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 [38]. If we imagine that such a profile will be applied to a component-based system, part of

---

2 Nancy Leveson says that when a safety engineer needs to assign a reliability to an embedded software component, it 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.

3 ‘Subdomain testing’ has also been called ‘partition testing.’
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 tools very efficient and may even allow tractable analysis of loops.

### 3 Testing-based Theory of System Synthesis

In physics, the science often taken as the ideal, understanding begins with microscopic theory. The physicist imagines low-level details of how a phenomenon might occur, describes those details mathematically, and works out how they explain the phenomenon. The kinetic theory of gases is a textbook example [28]. By imagining the molecules of a gas in elastic collision, in principle it is possible to calculate observable properties of the gas. The theory isn’t practical nor exactly correct, but it is a superb mental model for understanding gas behavior. In a way analogous to ideal gas theory, this section presents a quantitative theory that predicts system properties from component values.

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 proper-
ties with numerical (real) values; some security properties can also be incorporated with a bit more difficulty. In order to be concrete, the theory will be presented for program run time. Run time is used as the presentation vehicle because it is easiest to understand and makes the best description of a basic theory of component-property composition. No attempt is being made to capture real performance analysis. In practice, the issues of operating-system overhead, memory-reference locality, cache hits, etc., may dominate the actual performance of a system, but here we are concerned only with the fundamental issues of how components compose. Other variants of the theory are discussed in Section 7.

Section 3.1 describes the measurements to be made by a component developer. Section 3.2 gives the rule for synthesizing a system of two components in sequence that is the heart of the theory. Synthesis rules for conditional and iterative constructions are given in Sections 3.3 and 3.4.

3.1 Approximating and Measuring Component Properties

A component’s run time over input space $D$ is a function $T$ on $D$. Intuitively, $T(x)$ is the run time for the component when it executes on input $x$. Now if two components $C_1$ and $C_2$ have run-time functions $T_1$ and $T_2$, when they are placed in series so that the output of $C_1$ becomes the input to $C_2$, the combination run time $T$ is $T(x) = T_1(x) + T_2(x')$, where $x'$ is the output of $C_1$ on input $x$.

In testing, $T$ is sampled by executing the component. By a combination of structural analysis and empirical investigation, suppose that the developer of a component $C$ divides the input space of $C$ into a finite number of subdomains $S_1, S_2, ..., S_n$, and approximates $T$ as a step function that has 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. Similarly, functional values for the component can be approximated by constants $<v_i>_{i=1}^n$. 

13
By sampling the input space and observing a component’s behavior, its developer can obtain sub-domains and values $<v_i>_{i=1}^n$ and $<t_i>_{i=1}^n$. It is sometimes possible to choose subdomains such that the approximation is perfect because $T$ is a step function; see Section 7.2. However, functional behavior is seldom well approximated. If the functional behavior of a component is numerical, then it is appropriate to obtain the value $v_i$ by systematic sampling and averaging over $S_i$. The run time is numerical, so its average is appropriate. For non-numerical functional behavior there may be no defensible single value to choose. Often, multiple values suggest a better choice of subdomains. For example, it makes no sense to average values of a Boolean function, but it may be possible to choose subdomains so that the Boolean results within each subdomain are either all \textit{true} or all \textit{false}.

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.
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 values of the run-time property, as measured by developers for the components \( B \) and \( C \), and calculated for the composite system \( U \). Let the component subdomains be \( S^B_1, S^B_2, ..., S^B_n \) and \( S^C_1, S^C_2, ..., S^C_m \) respectively (usually \( n \neq m \)), and let their corresponding output-value vectors be \( <v^B_1, v^B_2, ..., v^B_n> \) and \( <v^C_1, v^C_2, ..., v^C_m> \). Let their run-time vectors be \( <t^B_1, t^B_2, ..., t^B_n> \) and \( <t^C_1, t^C_2, ..., t^C_m> \). It is desired to calculate a set of \( k \) subdomains for the system \( U \): \( S^U_1, S^U_2, ..., S^U_k \), and two corresponding step functions: \( <v^U_1, v^U_2, ..., v^U_k> \) for system output and \( <t^U_1, t^U_2, ..., t^U_k> \) for system run time.

The calculation defines \( U \) as an ‘equivalent component’ for the system of two components, 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 3.5). To calculate the equivalent component means finding a set of subdomains and
the input-output and run-time vectors for the configuration system.

In the series case, the system subdomains are those of the initial component $B$, so $k = n$ and
$S^U_i = S^B_i$, $1 \leq i \leq n$.

On subdomain $S^U_i$, $B$ has output $v^B_i$. Let this fall in the $j^{th}$ subdomain of the following com-
ponent $C$. Then the system output value on $S^U_i$ is $v^C_j$. That is:

$$v^U_i = v^C_j, \text{ where } v^B_i \in S^C_j.$$  \hspace{1cm} (1)

The runtime of the system on subdomain $S^U_i$ is the run time of $B$ there plus the run time for $C$
on $S^C_j$:

$$t^U_i = t^B_i + t^C_j, \text{ where } v^B_i \in S^C_j.$$  \hspace{1cm} (2)

Data vectors for the components that make up a series system allow the system vectors of
equations (1) and (2) to be calculated. Component data is provided by the component develop-
ners for each real component used in a system and it results from previous equivalent-component
calculations when the equations are applied repetitively as described in Section 3.5.

It is evident that on large, haphazardly-chosen subdomains the step-function approximation to
component behavior is a poor one. This is particularly true of the approximation to functional
behavior, since the properties of the calculated equivalent component for a system depend on the
way one component feeds into the next. For poor subdomains, this connection is complex—one
subdomain maps into several—but the step-function approximation cannot capture this complex-
ity. Nevertheless, the intuition behind the theory is that as the subdomains shrink in size, the
approximation should be better and the theoretical predictions should improve$^4$.

$^4$Another approach is to use a better approximation than step functions. See Section 5.
If and when the step-function approximations are perfect—that is, the actual functional and run-time values are constant on every subdomain\textsuperscript{5}, the predictions are exact.

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 exactly correct\textsuperscript{6}.

### 3.3 Conditional System Control Structure

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

\[
\text{if } B \text{ then } C_T \text{ else } C_F \text{ fi.}
\]

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

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 members of \( D_F \) reach \( C_F \). The

\textsuperscript{5}It is not enough to have perfect approximations on only some subdomain(s) of interest, as will become clear in the experiments of Section 6.

\textsuperscript{6}In early experiments with the theory, the input space was restricted to the domain of integers, thus forcing the granularity to 1. Experiments were run using exhaustive testing, and indeed it was found that in the singleton limit the predictions of the theory are perfect. There are technical difficulties in using an integer domain with the piecewise-linear approximations described in Section 5, so the experimental domains were changed to be double-precision floating-point values, with a granularity too small to explore directly.
subdomains of the equivalent component to be computed are therefore:

\[ D_T \cap S_{i}^{C_T}, 1 \leq i \leq n; \quad D_F \cap S_{j}^{C_F}, 1 \leq j \leq m. \]  

(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.

\( D_T \) and \( D_F \) are natural subdomains to use for the conditional-test component \( B \), because they exactly capture \( B \)'s input-output behavior. However, they may have to be further refined to capture the run-time behavior of \( B \). It makes no sense to give \( B \) defining subdomains that cross the \textit{true} – \textit{false} boundary (that is, \( S_{k}^{B} \cap D_T \neq \emptyset \land S_{k}^{B} \cap D_F \neq \emptyset \), for some \( 1 \leq k \leq p \)), which poses a practical barrier for test functions that change value frequently.

A conditional construction with no \textbf{else} part can be handled by taking \( C_F \) to be an identity component with zero run time.

### 3.4 Iterative System Control Structure

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

Begin by unrolling the loop

```
while B do C od to if B then C fi; while B do C od.
```

The trailing loop after the unrolled conditional is called the \textit{residual loop}. On any subdomain \( S_{i}^{C} \)
where $B(v_i^C)$ is false, the residual loop makes no contribution. 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 $B$ goes false on them. Continuing, at each unrolling at least one subdomain is eliminated, or none is eliminated. In the latter case, no further changes can occur, and 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.

If the $C$ subdomains are not fine enough to capture the 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 entirely 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 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.

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

<table>
<thead>
<tr>
<th>Construct</th>
<th>Polish</th>
</tr>
</thead>
<tbody>
<tr>
<td>X; Y</td>
<td>XY S</td>
</tr>
<tr>
<td>if Z then X else Y fi</td>
<td>ZXY C</td>
</tr>
<tr>
<td>while Z do X od</td>
<td>ZX L</td>
</tr>
</tbody>
</table>

For example, Fig. 3 shows the Polish representation for an illustrative flowchart and its reduction to an equivalent component (E4). In the figure, components are named by integers. Familiar algorithms could parse the flowchart in the figure or equivalent pseudocode to obtain the Polish but for the prototype tools described in Section 4 the Polish is hand generated.

A research prototype CAD tool for system design takes such a reverse-Polish system description along with a list of its component descriptions (subdomains, output- and run-time vectors) and uses the theory’s rules to construct each equivalent component in order. The final component (E4 in Fig. 3) has as its functional- and run-time vectors the calculated behavior of the complete system.

---

7 Although system synthesis was envisaged from the outset, in previous published versions of the theory [20] the constructions were not carried out past the first level, and there was a serious mistake that was discovered only when we implemented the CAD tools.
Figure 3: System synthesis by computing equivalent components

Historical Note

This research has been in progress for about four years [19, 16, 20, 21, 18]. Reference [21] gives details of the original theory (for reliability). As tools were developed to support the theory and experiments explored its workings, substantial changes were made. First, it was realized that the reliability application, though the most important, was not the easiest or best for experiments and exposition, so the theory was applied to run time and tools written for this application. Second, the original step-function approximation described in this section was augmented by a more accurate piecewise-linear approximation (Section 5 to follow) and the tools modified to support it also. Third, the initial decision to experiment with integer-valued functions on integer domains was
wise only for the step-function approximation, so tools were changed to use floating-point data.

In the original description of synthesizing component properties [20, 21] we did not appreciate the necessity of calculating an equivalent input-output behavior for composed components. We defined a ‘transfer matrix’ to describe how points in the subdomains of an initial component are mapped into the subdomains of a subsequent component and we made use of the actual input-output behavior of the initial component in measuring this matrix. A transfer matrix captures the input-output behavior of an initial component $B$ more accurately than does the step function $<v_1^B, v_2^B, \ldots, v_n^B>$, but it has the serious shortcoming that it depends on both components in a series combination, and thus has to be measured at system-design time, not at component-development time. Furthermore, the transfer-matrix idea cannot be used to synthesize an arbitrary system because the equivalent components that would result from one calculation do not have complete input-output behavior. They are defined only by the subdomain, which trivializes the transfer matrix.

Supporting tools and these changes in the theory are described only in this paper.

4 Tool Support

Research-prototype tools were implemented in Perl to test and specify components and to make the theoretical predictions described in Section 3. The latter is a rudimentary system-design CAD tool. A number of support tools were also written to analyze and display the results of measurement and synthesis. Altogether the tools comprise about 2700 lines of non-comment Perl code. The tools handle components with a single floating-point input and output, whose behavior uses no persistent state.
4.1 Component-specification Tools

A component developer has the executable code, and must create a catalogue entry. In this theory that amounts to a list of subdomains and a value for the functional- and run-time values on each subdomain. The subdomain list and the name of the executable file are recorded in a configuration file for the component, and a specification tool creates the test-based specification (catalogue entry) by attaching to each subdomain a pair of values (function, runtime). The step-function values are averages obtained by systematic sampling over the subdomain. A support tool produces a plot of the specification compared to the actual component behaviors with the measured standard deviation (root-mean-square error) for each subdomain. Figure 4 is an example of the tool output for a component whose behaviors are linear. The horizontal axis in Fig. 4 is marked with the subdomain boundaries, chosen to capture the behavior to about the same accuracy throughout the arbitrary interval of [0,10). The approximation is perfect only for functional values on the subdomain [4,5).

Information like Fig. 4 may be helpful to a developer in visualizing component behavior. However, tools like the prototype have the primary purpose of creating the catalogue description, not analyzing a component. The developer must have a specification and must perform whatever verification is needed to be satisfied that the component meets that specification. Then the tools are used to approximate the actual behavior and catalogue the approximation.

4.2 System-design (CAD) Tools

The CAD tool takes as input a collection of component specifications (produced by the specification tool of Section 4.1) and a description of the system into which they are to be combined.
### Table

<table>
<thead>
<tr>
<th>Subdomain</th>
<th>rms errors (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>interval</td>
<td>functional</td>
</tr>
<tr>
<td>[0.0, 0.2)</td>
<td>9.52</td>
</tr>
<tr>
<td>[0.2, 0.5)</td>
<td>7.41</td>
</tr>
<tr>
<td>[0.5, 0.8)</td>
<td>6.06</td>
</tr>
<tr>
<td>[0.8, 1.0)</td>
<td>5.13</td>
</tr>
<tr>
<td>[1.0, 1.2)</td>
<td>4.44</td>
</tr>
</tbody>
</table>

(8 similar rows omitted)

<table>
<thead>
<tr>
<th>Subdomain</th>
<th>rms errors (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[3.2, 3.5)</td>
<td>2.02</td>
</tr>
<tr>
<td>[3.5, 3.8)</td>
<td>1.90</td>
</tr>
<tr>
<td>[3.8, 4.0)</td>
<td>1.80</td>
</tr>
<tr>
<td>[4.0, 5.0)</td>
<td>0.00</td>
</tr>
<tr>
<td>[5.0, 5.6)</td>
<td>2.03</td>
</tr>
<tr>
<td>[5.6, 6.2)</td>
<td>2.17</td>
</tr>
<tr>
<td>[6.2, 6.9)</td>
<td>2.32</td>
</tr>
<tr>
<td>[6.9, 7.5)</td>
<td>2.49</td>
</tr>
<tr>
<td>[7.5, 8.1)</td>
<td>2.69</td>
</tr>
<tr>
<td>[8.1, 8.8)</td>
<td>2.93</td>
</tr>
<tr>
<td>[8.8, 9.4)</td>
<td>3.21</td>
</tr>
<tr>
<td>[9.4, 10.0)</td>
<td>3.55</td>
</tr>
</tbody>
</table>

Weighted average 2.9 2.2

**Figure 4**: Actual (dotted line) and approximated (solid line) behavior for a simple component, and measured approximation error
Its output is a system specification in the same form as the component specifications; that is, an ‘equivalent component’ for the system. The system description is given to the tool in the reverse Polish form of Section 3.5, and the CAD tool carries out the successive syntheses of equivalent components as in the example of Fig. 3, using the algorithms of Section 3. A support tool produces a plot of the predicted behavior. The predictions are compared with values obtained by running a system formed by linking together the actual component code in the given structure.

To illustrate these tools, the component shown in Fig. 4 was placed in the simple system of Fig. 3 for the components numbered 1, 3, 5, and 6. The test components 2 and 4 were taken to be a component that returns \textit{true} only in the interval \([2,6)\). When the CAD tool is run on this system, it synthesizes a series of equivalent components, corresponding to the subsystems that make up Fig. 3, which culminates in an equivalent component for the complete system. The first such subsystem consists of the loop, then the series combination of the loop subsystem with the initial component, and so on. In the example, the loop is unwound three times in calculating its equivalent component (E1 in Fig. 3) before the approximation terminates in every subdomain.

Figure 5 shows the resulting predictions with the actual system behavior systematically sampled for comparison. The first thing to notice about the system behavior is that even for this very simple expository example it is fairly complicated. There are three discontinuities, at inputs not simply related to inputs where the components are discontinuous. The step-function approximation captures the run time pretty well; the functional approximation is less good. Even when the approximation is relatively good, there are systematic errors—for example, following the discontinuous drop just past input 2, the functional approximation is consistently high, while the run-time approximation is consistently low. These and other features of system predictions will be explored.
Figure 5: Predicted (solid line) and actual (dotted line) behaviors for a simple system

in detail in Section 6.

The tools that execute the composite system for comparison with the theory’s predictions include a trace facility by subdomain. Each input given to the system is monitored to see which (if any) subdomains it reaches in each of the components that comprise the system. The trace data allows the construction of a profile that each component sees in place. Even the simplest system illustrates the profile dilemma raised in Section 2. For example, if the system of Fig. 3 is given inputs from a uniform profile, the profile seen by the component in the \textit{false} branch of the conditional (number 6 in Fig. 3) is shown in Fig. 6. There is no way that the component developer can know that this would be the appropriate test profile; using a uniform test profile for component 6 would be completely wrong.

The example of this section is contrived to show off the operation of the prototype tools. How-
Figure 6: Profile seen in place by component 6 on uniformly distributed system input

ever, it was not adjusted to exhibit the features that it does, features that appear in almost any case in which component behaviors have a substantial variation. The system behavior is always surprisingly complex, profiles in place are always seriously distorted, and the approximations are good but show some systematic errors. Section 4.4 further considers the use of contrived but demanding components in validation.

4.3 Test-based ‘Specifications’

There is another way to obtain system predictions that emphasizes the character of this synthesis theory. The component catalogue entries themselves could be ‘executed’ by table lookup. That is, to obtain the output of a component for input $x$, $x$ could be looked up in the subdomains of its specification and the constant value returned for the appropriate subdomain. If all component
specifications are linked together in the proper structure and ‘executed,’ the result duplicates the behavior of the equivalent component computed by the CAD tool. In a sense, the CAD calculations are ‘compiling’ the system into a single equivalent component that can be ‘executed’ by a single lookup of the same kind. Section 4.5 discusses the performance of the tools, and compares the times required to execute a real system, to execute a system by table lookup in each component specification, and to make the CAD calculation. In this section, the unusual sense of these component ‘specifications’ is examined.

An engineering component catalogue contains specifications for its entries, descriptions of their properties to be used in selection and in system design. However, for software, ‘specification’ carries a somewhat different connotation 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 the whole 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 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 ‘specification’ occurs in recent work by Ernst on the Daikon system[40] and Henkel [22]. 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 our catalogue descriptions, but they are in an entirely different form and obtained by quite different means. For us, 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. For example, Daikon generates only empty pre- and post-conditions for the examples in this paper because it does not attempt to capture explicit functionality beyond the linear. (For linear functions, its ‘specifications’ exactly match ours.) In a personal communication, Henkel indicated that he believes his system would give similar vacuous results on our examples. Neither mathematical ‘specification’ system has been applied to non-functional properties.

4.4 Components Constructed to Order

The components employed in this example and in other experiments reported in Section 6 are artificial. Initially, we tried to create or find ‘real’ components with behaviors that would reveal the workings of the theory, with unsatisfactory results. We first used a vending-machine specification that is a standard object-oriented programming exercise. The components of the machine select a product to vend, accept money deposited, make change, and so on. A Java implementation was compared to the reliability application of the theory with excellent results. However, closer examination showed that this was a trivial consequence of the simplicity of the code: the behaviors were constant on obvious subdomains so the example was no real test of the theory. Secondly, we crafted a Java component that calculated by brute-force trial the spacing between the two successive prime
numbers larger than its input, and placed this in series with another component that performed a number of time-wasting operations proportional to its input value. The series system thus had a composite run time that increased with input in a complex way. At first the theory results were wildly inaccurate, which we traced to measurement errors when the Java run-time environment did a garbage collection; we also had difficulty with repeatability of the system measurements on UNIX systems where the accuracy of the timing primitives is limited to one scheduling tick. When these problems were sorted out, the results were again excellent, but again less significant than the effort that went into obtaining them.

Because the prime example was long-running when repeatedly executed enough to overcome the scheduling measurement errors, we hit upon the idea of constructing ‘phony’ components to order. Any executable program can be surrounded by a wrapper that measures properties of its execution. In particular, we had been obtaining process run time by an operating-system call in the wrapper, sending the run-time value to \texttt{stderr}. In retrospect it is an obvious insight to realize that we did not have to wait for a component to execute, but could simulate its behavior. If a phony component is created that simply writes run-time values to \texttt{stderr} but does not actually use that time, one cannot tell the difference between it and a real component with a timing wrapper. This trick has several advantages:

1. It allows us to create arbitrary component run-time functions.

2. It greatly speeds up run-time experiments.

3. It eliminates fluctuations in run times caused by operating-system timing.

The first of these is the most important. The arbitrary functions could represent other non-functional
properties such as reliability. They can be made arbitrarily complex in a direct way, without the need to consider what a component is ‘really’ doing.

Closed-form formulas were used in phony components to produce interesting run-time functions. For example, the following is the Perl code for the component whose behavior is plotted in Fig. 4:

```
$x = <STDIN>;

if ($x < 4) { $y = 2*$x + 1.5; }
elsif ($x < 5) { $y = 9.5; }
else { $y = -.9*(x-5)+9.5; }
$t = .1*$x + .3;

print "$y
"
print STDERR "$t
";
```

A generator was also written to create a phony component that has the behaviors of a pair of given finite graphs with linear interpolation between the points. It allows a ‘real’ component to be replaced with an efficient phony one by sampling the real component’s properties and giving the graphs to the generator. We tested the generator by applying it to the previously mentioned prime component, where we were able to reproduce the results exactly, including the discontinuities resulting from the Java garbage collection.

Testing methods have traditionally been validated using a small collection of ‘toy’ programs\(^8\). The constructed components employed here can be thought of as worse than ‘toys.’ However, our example components are harder on the theory than real ones would be: they are contrived

---

\(^8\)The triangle-classification program first employed by Glenford Myers [39] is perhaps the most used. One journal referee is reported to have stated that she would reject out of hand any submission that employed this program in a validation study.
precisely to explore it as real components failed to do in our initial attempts. The component used in the exposition of Section 4.2, for example, has input-output behavior that varies widely and non-monotonically over its domain.

4.5 Performance of Analysis and Synthesis Tools

Although little effort was expended on efficiency in the prototype tools, their performance is promising.

The analysis tools for component developers necessarily use brute-force sampling, so their running time is proportional to the number of subdomains, the number of test samples in each, and the component execution time. We tend to discount the inefficiency of these tools, because a component developer is doing the measurement/specification work once, to be used by all subsequent systems designers.

The performance of CAD synthesis tools is of greater interest, because their efficiency determines how long a systems designer has to wait for predictions of a particular design and hence how easy it is to try different designs.

The synthesis time for a conditional is roughly proportional to the number of subdomains $K$. For a sequence the synthesis time is proportional to $K \log K$, since for each subdomain of the first component, a range subdomain in the second component’s input domain must be found by binary search. A loop requires at a minimum the time for a conditional synthesis and one sequence synthesis for each time the loop must be unwound. (The number of subdomains that must be processed in the sequence decreases with each unwinding.) To make a rough calculation, suppose that in the system to be synthesized the number of conditionals and sequences are about the same,
that each loop requires just one iteration, and that the number of loops is small compared to the number of other constructs. The synthesis time for a system of $M$ constructs (roughly the same as the number of components) with $K$ subdomains each is thus proportional to $\frac{MK}{2}(1 + \log K)$. On a 2 Ghz PC the synthesis time for the example system, with $M = 5, K = 256$, is about 240 ms, so the proportionality constant is about 0.04 ms. For a system with 100 components and 1000 subdomains the synthesis time would be about 20 sec.

The testing alternative to synthesis calculations is to actually execute the assembled system. Actual system execution time is the product of the number of subdomains, the number of samples taken in each, and the execution time per sample$^9$. If in the example of 100 components and 1000 subdomains the average run time of the system were 100 ms and 20 samples were taken per subdomain, such a system would take 2000 sec, more than a half hour, to test directly. Thus CAD tools are more practical than executing a composite system$^{10}$.

An intermediate solution to the measurement of system performance is to assemble the component specifications into a system that can be executed by table lookup as described in Section 4.3. This table-lookup system has a run time that does not depend on the actual component run times, but only on the number of components and the number of subdomains, since its executions amount to looking up the output for each subdomain in sequence through the components. Thus the lookup system executes in a time proportional to $MK \log K$. For the example system with $M = 5, K = 256$, the time$^{11}$ is about 12 sec, so the proportionality constant is about 1.2 ms, and

---

$^9$Testing an actual system need not use subdomains, but for comparison with the CAD calculations total samples can be represented by the product of subdomain count and samples/subdomain.

$^{10}$The time to assemble, debug, and test the composite is ignored, but in practice would probably dominate the execution time, making the case for theoretical calculation even stronger.

$^{11}$The inefficiencies of the prototype tools are particularly apparent in this measurement.
for 100 components and 1000 subdomains the lookup run time would be about 20 minutes.

If the system run time per test sample rises, it will be because components have longer run times, but the CAD calculations (and also execution of the system formed from the specifications) are not more expensive for longer-running components. In the example, if the average system run time were 200 ms, the CAD tool would still take 20 sec, running the table-lookup system would still take 20 minutes, but testing the actual system would take more than an hour.

It is unrealistic to assume that all loop constructs terminate quickly. In realistic cases, looping could make synthesis impractical. However, actually executing loops to determine system behavior would be equally impractical. There is no efficient way to handle large iteration counts in any testing-based approach.

5 Linear Approximation

When a component’s output varies, it may range over diverse subdomains of a following component, and to ignore this variation as the step-function approximation does misrepresents the interaction. The simplest better approximation is linear functions in place of constant values: a piecewise-linear approximation.

The theory for piecewise-linear approximation is not very different from that given for step-function approximation in Sections 3.2 (sequence) and 3.3 (conditional). (There are no changes at all to the synthesis of loops or to system synthesis, since these are cast in terms of the other two constructions.) Instead of a vector of values, e.g., \( \langle v_1^B, v_2^B, ..., v_n^B \rangle \) approximating functional output values on the \( n \) subdomains of component \( B \), the piecewise-linear approximation replaces each \( v_i \) with a pair (slope, intercept) defining a line. Instead of measuring an average value over
the subdomain, the component developer finds the best-fit line.

A conditional construction is synthesized for linear approximations exactly as in Section 3.3, by intersecting subdomains and passing on to the equivalent component the true or false component’s functional value, which is now a linear function described by a (slope, intercept) pair. For the composite run time, the linear function approximating the test component and the one approximating the proper following component are added to get the equivalent-component linear function.

The series construction is a bit more difficult to synthesize for linear approximations. In the notation of Section 3.2, consider one subdomain interval $S_i^B = [L, R]$ of the first component, in which the functional behavior is described 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 $S_i^B$). Figure 7 illustrates this subdomain construction. The vertical heavy line is a $C$ subdomain and the horizontal heavy line is its reflection into the $B$ subdomains, which is a subdomain of the calculated equivalent component. 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 runtime functions for the components (say these are: $\lambda x(hx + r)$ and $\lambda x(h'x + r')$), with the second

\[\text{The derivation is correct only for slope } k > 0. \text{ When } k < 0, \text{ the end points of the interval in the second component’s domain reverse, and there is a technical difficulty because the right end of the interval is open.}\]
adjusted to receive an input that is the functional output of the first component. This sum line has slope $h + kh'$ and intercept $r + h'q + r'$.

Repeating this calculation for the intersections between each $S^C_j$ and the output ranges of each $S^B_i$ results in a list of subdomains and linear functions on them for the composite functional and run-time behavior\(^\text{13}\).

The theory for step-function approximation is just a special case of the linear-function approximation in which all lines have slope zero. Perfect predictions result when all behaviors are actually linear in every subdomain.

Piecewise linear approximation has two important technical advantages over the simpler step-function approximation. First, it can perfectly represent the identity function, which is used in synthesizing a conditional with no \textbf{else}, in particular as a part of the synthesis of a loop. Second, linear approximation vastly improves the synthesis of the series construction. With step functions,\(^\text{13}\)

\(^\text{13}\)The technical problem that forced us to abandon the simpler integer subdomains is that the ‘gaps’ of size 1 between integer intervals in a second component, when reflected back to form new subdomains in the composite equivalent component, may leave arbitrarily large gaps there.
when two components are placed in series the composite system-equivalent component has the same subdomains as the first component in the series and so information about the second component’s subdomains are lost. But with the piecewise-linear approximation, whenever a linear output from the first component crosses a subdomain boundary in the second, the system equivalent component acquires a subdomain boundary there. The effect of this is to propagate boundaries that isolate changes in behavior.

It is natural to imagine more accurate approximations to behavior, say with higher-order polynomials. There are two reasons why this was not explored: (1) As will appear in the experiments of Section 6, the accuracy on a subdomain seems to be less important than the choice of subdomain boundaries. Discontinuities that fall within subdomains are a fact of life, and no approximation can do very well with them. (2) Piecewise-linear functions form the highest-order class that is closed under composition. Two linear functions compose into a linear function; two quadratics in general compose to a quartic, etc.

5.1 Tools for Piecewise-linear Approximation

The changes required to add linear behavior approximations to the tools described in Section 4 are mostly straightforward. In every case where the tools used a constant value on a subdomain, the equation of a line is used instead. For example, to obtain a component’s output on input \( x \) within subdomain \( S \), instead of looking up a constant value for \( S \), the line equation for \( S \) is looked up and evaluated at \( x \).

The component-specification tools must now find and record line equations for each subdomain. The line can be obtained as a best fit (in the sense of least squares of deviations), or by
simply using the values at the interval end points. As subdomains shrink, the difference between these alternatives should decrease. In the work reported here, the lines are obtained by best fit, partly because we are interested in the approximation error which is thereby minimized.

The most difficult changes are those required in the CAD tool for the case of series composition, because the piecewise-linear approximation requires calculation of new subdomains for the equivalent component. There are a number of tricky special cases that arise in determining their boundaries.

When the modified tools are run on the example of Section 4.2 in which the actual behaviors are piecewise-linear, the approximation is perfect, and the system predictions agree exactly with measurements.

6 Revealing Experiments

Although physics is our model for theoretical explanations, experimentation with software theories differs in principle from experiments to test a scientific theory. In science there is an objective reality, sometimes called ‘nature,’ which determines the possible experiments. Nature also simplifies the experimental situation because many natural phenomena are continuous. To the contrary, software is a human creation that can be changed at will, and lacks simplifying continuity. A software theory is useful if it captures and simplifies software properties so that they can be understood. Experimentation with a theory seeks to observe real software behavior, and to learn if the theory exhibits the same behavior but is easier to control and understand. ‘Validation,’ checking the correspondence between theory and reality, is not as meaningful for software as for physics. As a human creation, software fails to be understood only because a particular example is too com-
plicated to grasp. A theory simplifies, and thereby distorts reality, but although this necessarily makes it ‘invalid,’ its explanations can be useful. We do not need to experiment with a further abstraction to learn about software—we can experiment with the thing itself. But the theory may be comprehensible where the software is not.

In software experiments there is a unique potential for error: the experiments themselves use software tools that can be faulty.

To be useful, an experiment comparing theory and reality must be ‘revealing.’ It should expose how well reality is captured, but in a way that teaches us something new about what is happening in the situation.

Experimental validation of basic software theory then has three purposes:

**Check the mathematics.** There can be a mistake in the mathematical theory, some important aspect of the situation improperly captured by a definition, or a proof in error. It is an important role of experiments to expose such mistakes\(^\text{14}\).

**Test the implementation of tools.** For simple cases it is possible to calculate the theoretical results by hand, and cases of perfect approximation can be tried. Such cases can be used to check tools that perform the calculations.

**Investigate the theory’s assumptions quantitatively.** Any theory fails to the extent that its assumptions do not hold. Learning how these failures manifest themselves and how the assumptions can be quantified to control inaccuracy is essential to improving a theory.

The final purpose is the most important. Experiments that serve it are revealing in the best sense.

\(^{14}\text{The use of a ‘transfer matrix’ in the initial theory [20] was such a definitional mistake.}\)
6.1 Systematic Subdomain-refinement Experiments

One way to investigate the theory is to run case studies in which subdomains covering part of the input space for a simple system are refined. In such an experiment we hope to see the accuracy of the theoretical predictions improve and stabilize. When the approximation is perfectly accurate, the predictions should be perfect. Aside from the fun of finding and fixing bugs in tools, the most interesting aspect of a case study is the insight it provides into the character of ‘bad’ subdomains. How large and ill-chosen can subdomains be, yet the theory still make relatively accurate predictions? What characteristics of subdomains influence the predictions?

A number of simple system structures were investigated using constructed components whose behaviors varied widely. The most complex had 12 components combined in nine system constructs with a structure including common patterns like conditionals within loops, sequences within conditionals, etc. The simplest systems had a pair of components in sequence, since the series construction is basic to the theory. Beginning with large subdomains that make little attempt to capture the component behaviors accurately, the subdomains were systematically refined to observe the improvement in prediction accuracy. As special cases, systems were constructed in which the approximations are perfect, to check that the tools then yield perfect predictions.

Similar results were obtained in all the experiments conducted, some of which were already seen in the expository example of Section 4. In summary:

- Surprisingly complex system behaviors arise from simple component behaviors.
- When the approximation is perfect (for example, in the expository example of Section 4 with the piecewise-linear approximation) the predictions are perfect.
• It isn’t enough for the functional approximation to be perfect for some components or for some subdomains; in general there must be no approximation errors anywhere for perfect results.

• Functional-value prediction dominates run-time prediction. When the component functions are predicted accurately, the run-time results behave nicely, improving in direct proportion to the accuracy of the component approximations of run time.

A simple but typical case study uses the structure of Fig. 3, but with complex component behaviors given in Table 1. Component 1 is the one shown in Fig. 4. Figure 8 shows a measured

<table>
<thead>
<tr>
<th>Component</th>
<th>Function ( y = f(x) )</th>
<th>Run time ( T = g(x) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( y = \begin{cases} 2x + 1.5 &amp; x &lt; 4 \ 9.5 &amp; 4 \leq x &lt; 5 \ -0.9(x - 5) + 9.5 &amp; x \geq 5 \end{cases} )</td>
<td>( T = 0.1x + 0.3 )</td>
</tr>
<tr>
<td>2</td>
<td>( y = true ) if ( 3 \leq x &lt; 5 )</td>
<td>( T = 0.2 )</td>
</tr>
<tr>
<td>3</td>
<td>( y = 0.2(x - 6)^2 + 1 )</td>
<td>( T = 0.1 )</td>
</tr>
<tr>
<td>4</td>
<td>( y = true ) if ( 1 \leq x &lt; 2 \lor 5 \leq x &lt; 6 \lor x \geq 8 )</td>
<td>( T = 0.3 )</td>
</tr>
<tr>
<td>5</td>
<td>( y = \begin{cases} 2 \sin x + 5.2 &amp; x &lt; 5 \ 3 \cos^2 x + 4 &amp; x \geq 5 \end{cases} )</td>
<td>( T = 0.6 - 0.04x )</td>
</tr>
<tr>
<td>6</td>
<td>( y = 7\cos(x^2/8)</td>
<td>e^{-x/7} + x/2 )</td>
</tr>
</tbody>
</table>

Table 1: Component behaviors in the system 1 2 3 L S 4 5 6 C S (Fig. 3)

specification of another component (6 in Table 1) using the piecewise-linear approximation with 20 subdomains. For component 6 the run-time approximation is perfect; Fig. 8 shows that the
Figure 8: Measured (solid line) and piecewise-linear approximate (dotted line) behavior of component 6.

Functional approximation is not bad except in a few subdomains (e.g., around input 6). An input domain of [0,10) was arbitrarily selected for the case study.

The corresponding measured system behaviors are shown in Fig. 9.

This example could hardly be more explicit and concrete: every detail of the component behaviors and system structure is given in Table 1 and Fig. 3. However, the very detail of a concrete example makes it intuitively unsatisfying; it is hard to grasp its significance. This unfortunate situation is characteristic of basic, quantitative models: their most intuitive expression is abstract. To return to the parallel with Turing machines, one does not study large examples to understand the idea.

Table 2 summarizes experiments on component behavior using the step-function approximation.
Figure 9: Measured behavior of the case-study system

as subdomains are refined. In the table, accuracy measures that average over the whole domain
(“Overall” and “Mean” columns) improve steadily as the subdomains are refined, so that by the
time there are 64 subdomains the overall errors in the functional values are about 4% and the
run-time errors are under 1%. The system prediction errors follow the component approximation
errors: at the bottom of the table, halving the subdomain size reduces the approximation error by
about half and the prediction error is similarly reduced. However, the measures that look at each
subdomain (“Max”) show large errors persisting and even increasing in a few subdomains. This
anomaly will be discussed below.

Using the piecewise-linear approximation gives better predictions, as displayed in Table 3. The
successive lines in Tables 2 and 3 represent the same subdomains in each component approxima-
tion. The final count of system subdomains in corresponding rows is larger in Table 3 because the
<table>
<thead>
<tr>
<th>Subdomain count</th>
<th>Average rms % error</th>
<th>Functional % error</th>
<th>Run-time % error</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Functional</td>
<td>Run-time</td>
<td>Overall</td>
</tr>
<tr>
<td>4</td>
<td>12.88</td>
<td>6.95</td>
<td>20.93</td>
</tr>
<tr>
<td>8</td>
<td>7.48</td>
<td>3.55</td>
<td>17.36</td>
</tr>
<tr>
<td>16</td>
<td>4.13</td>
<td>1.77</td>
<td>15.32</td>
</tr>
<tr>
<td>32</td>
<td>2.07</td>
<td>0.90</td>
<td>7.47</td>
</tr>
<tr>
<td>64</td>
<td>1.05</td>
<td>0.45</td>
<td>4.29</td>
</tr>
<tr>
<td>128</td>
<td>0.52</td>
<td>0.22</td>
<td>2.01</td>
</tr>
<tr>
<td>256</td>
<td>0.27</td>
<td>0.10</td>
<td>1.15</td>
</tr>
<tr>
<td>512</td>
<td>0.15</td>
<td>0.05</td>
<td>0.73</td>
</tr>
<tr>
<td>1024</td>
<td>0.07</td>
<td>0.05</td>
<td>0.21</td>
</tr>
</tbody>
</table>

**KEY:** “Average rms error” is the root-mean-square deviation of the approximation averaged across all weighted subdomains and all components. In the groups of four columns headed “Functional %error” and “Run-time %error”, “Overall” error is a direct comparison between prediction and system measurement obtained by averaging equispaced samples across the input domain without regard for subdomain boundaries; “Max” is the largest error over all subdomains; “Mean” is the average over all subdomains; “>5” is the number of subdomains with error of more than 5%.

Table 2: Prediction accuracy as subdomains are refined (step-function approximation)
### Table 3: Prediction accuracy as subdomains are refined (piecewise-linear approximation)

<table>
<thead>
<tr>
<th>Subdomain count</th>
<th>Average rms % error</th>
<th>Functional % error</th>
<th>Run-time % error</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Overall</td>
<td>Max</td>
</tr>
<tr>
<td>13</td>
<td>3.78</td>
<td>10.52</td>
<td>9.62</td>
</tr>
<tr>
<td>26</td>
<td>1.42</td>
<td>3.28</td>
<td>5.94</td>
</tr>
<tr>
<td>52</td>
<td>0.33</td>
<td>1.18</td>
<td>7.97</td>
</tr>
<tr>
<td>96</td>
<td>0.12</td>
<td>0.44</td>
<td>1.66</td>
</tr>
<tr>
<td>191</td>
<td>0.02</td>
<td>0.06</td>
<td>0.22</td>
</tr>
<tr>
<td>384</td>
<td>0.00</td>
<td>0.01</td>
<td>0.04</td>
</tr>
<tr>
<td>775</td>
<td>0.00</td>
<td>0.01</td>
<td>0.01</td>
</tr>
</tbody>
</table>

**Comparison between step-function and piecewise-linear approximations**

<table>
<thead>
<tr>
<th></th>
<th>Functional</th>
<th>Run-time</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>0.52</td>
<td>0.22</td>
<td>2.01</td>
<td>4.41</td>
<td>0.88</td>
<td>0</td>
<td>0.43</td>
<td>7.78</td>
<td>0.20</td>
</tr>
<tr>
<td>96</td>
<td>0.12</td>
<td>0.00</td>
<td>0.44</td>
<td>1.66</td>
<td>0.13</td>
<td>0</td>
<td>0.00</td>
<td>0.02</td>
<td>0.00</td>
</tr>
</tbody>
</table>

The piecewise-linear algorithm creates new subdomains for each series synthesis. That these subdomains are an improvement over the step-function case is illustrated by the final two rows in Table 3, which repeat the 6th and 4th rows from the tables respectively. Although its subdomains are about four times as large for the components and 30% fewer for the system, the piecewise-linear predictions are about four times more accurate. Table 3 also displays the anomaly of persistent errors in a few subdomains in the first three rows.

There is a nearly linear relationship between the approximation error in the component mea-
surements and the prediction error, shown in Figure 10. The proportionality constant is 2.7 for the piecewise-linear approximation, 3.7 for the step-function approximation; for the latter, the first two rows of Table 2 have been omitted as outliers. The proportionality constant also grows with the complexity of the system being synthesized. This is the expected behavior, suggestive for a future theoretical error analysis.

![Figure 10: Prediction error as a function of approximation error (upper curve: step-function; lower: piecewise-linear)](image)

The information in Tables 2 and 3 is better displayed graphically. Figure 11 shows the system predictions for the step-function approximation (128 system subdomains); Figure 12 is for the piecewise-linear approximation (96 subdomains). The superiority of the piecewise-linear approximation is evident. System subdomains created by the piecewise-linear synthesis algorithm are fewer, but better capture the behavior.
Figure 11: Step-function-approximation predictions (solid line) and measurements (dotted line)

Figure 12: Piecewise-linear-approximation predictions (solid line) and measurements (dotted line)
Figure 13 displays a small region of Fig. 11, showing a subdomain in which inaccuracy persists—the anomalous behavior in Tables 2 and 3. The explanation is that the predicted behavior can only change at subdomain boundaries. If there is a rapid change in actual behavior (a discontinuity in the figure) within a subdomain, the prediction cannot track it unless it falls near a subdomain boundary. Halving the subdomain size only recreates the same (or worse) error in a new subdomain. However, such errors are confined to subdomains that occupy a smaller and smaller part of the whole domain, so the overall errors decrease. There is nothing the component developer can do to eliminate these troublesome system subdomains, because the points of discontinuity or rapid change in the system behavior are only determined by the combination of components—that is, they are emergent system properties. The component developer can only make a good approximation and hope for the best. It would be possible to identify potential poor
system subdomains and adjust their boundaries as a part of the synthesis calculation, but this does not help. A boundary created after the fact cannot affect the quality of the measured component approximations.

7 Discussion: Theoretical Issues

We have presented a quantitative description of component synthesis based on testing, for functional and non-functional properties. The work is more general than the run-time example used to present it, and this section explores its application to other non-functional properties and some issues common to all applications.

7.1 Combining Tested Components with Correct Ones

Although mathematical functional analysis of programs is in principle a solution to the problem of calculating any system property $T$ from component properties, the methods required are those of formal program verification. Exact system calculations would require information equivalent to a correctness proof of each component [37] as well as analysis of its $T$. 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 we 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.

For functional behavior, in equation (1) of Section 3.2, instead of finding $j$ by where $v_i$ falls, use $V(v_i)$:

$$v_i = v_j, \quad \text{where } V(v_i) \in S_j,$$

(4)

For run time, replace equation (2) with:

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

(5)

(Similarly change equation (6) for reliability, but here there is no additional factor introduced for $T(v_i^B)$ because reliability for the proved component has value 1.)

Thus component $V$ is just extending the functionality of $B$.

It is disappointing that for the piecewise-linear approximation similar general constructions do not work; the equivalent components that might be produced do not have linear behavior unless $V$ is linear. Given $V$, the correct component could be given a set of subdomains and a piecewise-linear approximation to $V$ to make the analysis work.

Although correctness proofs are not generally accepted as a practical method of software analysis, for the serious component developer this position may need rethinking. The restricted size of components and the existence of good specifications may make proof an attractive alternative to testing. Two additional techniques can replace proof in some situations:

**Exhaustive testing.** A component with a finite input domain can be exhaustively tested to exactly determine its functional- and non-functional behavior\(^\text{15}\). Knight [29] has suggested that this

\(^{15}\)It is easy to forget that part of the job is to be sure that unexpected inputs do not cause trouble outside the exhaustively tested domain. For this, informal proof techniques are appropriate.
kind of design is possible more often than one might think.

*Self-checking.* Manuel Blum [3] and Ammann and Knight [2] have independently suggested that some programs can be made to perform random redundancy checks at run time, which are sufficient to estimate a component’s reliability. Although this reliability is not 1.0, it is a profile-independent bound. There is no evident application of this technique to properties other than reliability.

### 7.2 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. In the experiments of Section 6 subdomains are systematically chosen as numerical intervals.

The insight that led testing theorists to look at subdomains in the first place [26] 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\(^\text{16}\).

\[^{16}\text{Mason [33] notes that care must be taken with potential run-time errors for this to be strictly correct.}\]

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 represent what a program
should do, not what in fact it does do.

Unfortunately, experiments show that it is more important to accurately approximate a component’s functional behavior than to capture its non-functional behavior. Here subdomain-testing experience is no help, since “same” subdomains for input-output behavior have never been investigated. In particular, neither path subdomains nor specification-based functional subdomains 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\textsuperscript{17}. 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.’

\textsuperscript{17}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 [16].
7.3 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 [15] and in reference [20] the first version of this theory was presented for reliability alone.

For reliability, the \( <t_i> \) vectors of Section 3 are reliability probabilities rather than run times. The component developer can estimate \( <t_i> \) by randomly sampling each subdomain and counting the fraction of failures. In the common case that no failures are observed in subdomain \( i \), \( t_i \) is a lower bound on the reliability there. Equation (2) becomes:

\[
t_i^U = t_i^B t_j^C, \quad \text{where} \quad v_i^B \in s_j^C.
\]

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

It is usual when discussing subdomain testing for reliability to assume that there is a constant failure rate (and hence constant reliability) in each subdomain. With this assumption, reliability experiments would take the same form as run-time experiments reported in Section 6. It makes no sense to consider a linear form for subdomain reliability.

Successful validation experiments on the reliability property are not convincing when no failures are observed in component testing, because the reliability bounds are not tight and the system calculations yield a loose bound that can fall within measured limits even if the theory is incorrect. To gain tighter bounds would require too many subdomains and too much sampling to be practical [6].

However, the run-time non-functional property was chosen instead of reliability to present the theory of Section 3 only partly because it is intuitively easier to understand. The deeper reason why reliability synthesis is difficult concerns 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 their run-time values are as given. 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 reports out a reliability different from 1.0, its functional values must be wrong no matter what subdomains are taken, and so the equivalent components and equation (6) will be in error.

Another way to describe the reliability case is to note that equation (6) 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 measured 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 very low failure rates of the first component $B$, it is unlikely that the choice of $S_j^C$ is wrong, hence equation (6) is very likely to be correct—that is, the components appear independent in the theory.

Finally, the difficulty of loop synthesis (Section 3.4) may be eased for a reliability calculation. The residual loop might be shown to make only a small contribution to the failure rate, in which case the analysis can be terminated without discovering that the loop terminates\textsuperscript{18}.

Superficially, the theory presented here applies to reliability without change; a closer look reveals the usual dubious nature of software reliability theory.

\textsuperscript{18} The predictions are suspect because an unverified hypothesis of termination is added: “if the loop terminates (in this subdomain) then the system reliability will be...” And of course if the loop does not terminate, then the system fails and the reliability is zero (in that subdomain).
7.4 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 or safety; software security is defined as the probability that a security assertion is preserved. 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 7.3. If the security assertions do not involve functionality, the independence problem does not arise.

Sometimes security or safety properties only emerge at the system level, however. Neither of two components has the property, which is expected to emerge from their combination. 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—it is not present at the component level. This case requires that testing a component in isolation measure as its memory-leak property the size and location of the net memory allocated, and that the systems calculation combine these allocations subdomain by subdomain to show that the composite is leak free.

Formally, let \( M \) be the set of all memory addresses. The non-functional values \( t_i \) are then 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: \( a_i \cap d_i = \emptyset \). A composition operator\(^{19}\) \( \oplus \) for components \( B \) and \( C \) forming a series system \( U \), corresponding to equation (2) for run time is:

\[
t^U_i = t^B_i \oplus t^C_i, \text{ where } v^B_i \in S^C_j
\]

\[
= (a^B_i, d^B_i) \oplus (a^C_j, d^C_j)
\]

\[
= (a^B_i \setminus d^C_j) \cup a^C_j, d^B_i \cup (d^C_j \setminus a^B_i))
\]

\(^{\text{Equation (7)}}\) (\(\setminus\) 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 usual efficient way of allocating memory is to get and release relatively large blocks infrequently. This pattern is fortunately easy to approximate accurately with few subdomains.

In the security application, a shift from proofs of security properties to more uncertain testing for those properties 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, the ability to test for such properties and to use a CAD tool to easily get approximate estimates of their system values has merit. First, testing provides a rough check on a proof. Everyone knows of supposedly water-tight proofs that turned out to be fallacious. Second, testing quickly catches gross mistakes that occur early in system design. Even if almost every system input violates security it can be difficult to discover this by proof methods. Testing methods find such failures immediately.

\(^{19}\)Equation (7) 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 nice compromise between sampling (testing) and proof appears in recent work by Jackson on the Alloy system [27] 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. It is surprising how often such a test exposes significant flaws in a logical specification.

7.5 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, 14] between subdomain testing and random testing (disregarding subdomain boundaries) have not shown any conclusive advantage either way in detecting failures. The experiments of Section 6 suggest an explanation.

Subdomain testing is most often used at the unit level, which is analogous to component-development time in CBSD. Furthermore, the subdomain boundaries in unit-test are selected in a somewhat haphazard way, only a few subdomains are used, 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 experiments of Section 6 suggest that the results would be very inaccurate. Failure to take care with subdomain boundaries and the paucity of subdomains and samples mean that as a unit specification, common-practice unit testing is almost worthless. If we expect to predict anything at system level, quite a different order of effort and care would be required.

The XP methodology presents an even more striking example. In XP, testing from customer scenarios, perhaps with only a few use cases, serves as the only real specification for each incre-
ment of the software being developed. These ‘specifications’ are induced in the way indicated in Section 4.3, and thus require far more than a few cases.

Our results suggest that testing using subdomains and to induce an approximate specification are good ideas, but that research is badly needed on what works and why, with the probable result that current practice must be much improved to have significant value.

8 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 [1] 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.
8.1 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 [23, 24], Guttag [13] and others [10, 11], Mills [37, 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 can then be described as follows:

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.

The most recent incarnation of the logic-based theories of behavior is based on Bertrand Meyer’s ‘design by contract’ [36]. 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, 43, 45].
8.2 Testing-based Models

Rosenblum [44] 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.

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 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. [46] 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 mea-
sured 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.

8.3 Summary of Related Work

The 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 our 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 7.4 and 4.3.
9 Future Work

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

9.1 Non-numerical Input/Output Types

The experiments and tools presented here are restricted to numeric input/output data types in components’ functional behavior. For non-numeric types, even so simple a one as ‘string,’ it is hard to imagine meaningful subdomains. However, the difficulty in defining subdomains where behavior functions are close to constant may be irrelevant. It may be that entirely wrong functional values can still lead to accurate non-functional predictions. For example, a component returning a character value may be ‘approximated’ as returning an ‘average’ of ‘m’ when this is not really meaningful, yet the run-time may be nearly the same no matter what character it returns and subsequent components’ run time may be similarly insensitive to the particular character they receive. Experiments to learn more about this case can be conducted with the existing tools by converting non-numeric values into floating point codes.

The hard part of such experiments is in the selection of experimental components. One can easily devise examples where the theory works very well, and others where it fails badly. What is a ‘representative’ example?

9.2 Theoretical Error Analysis

The experiments in Section 6 show that the synthesis theory is able to make pretty good predictions using a reasonable number of subdomains, and that the prediction error decreases as the approximation of component behavior improves. However, no theoretical quantitative relationship has
been established between approximation and prediction. A precise error analysis is needed so that the CAD synthesis tools can bound the error that may occur in their calculations. A component developer cannot know how accurately each component must be specified—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 there is a precise error analysis.

The ingredients of a proper error analysis are evident from the results of Section 6: things go wrong when the component behavior (particularly its functional behavior) is rapidly changing. Where this occurs, subdomains must shrink. It should be possible to bound the error made in synthesizing each system construct in terms of the approximation error of the components involved.

An error analysis for the reliability application (Section 7.3) 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 [49] have devised a reliability theory that includes a measure of the functional error, which may be just what is needed for analysis.

9.3 The Problem of State

Persistent state is a necessary feature of useful components that is not captured by the theory presented here. Software reliability engineering handles internal state by making state variables ‘hidden inputs’ and adding them to the input space for random sampling. This treatment is simply
wrong: state variables are not independent of the other inputs because their values are created by
the software itself.

A preliminary investigation has been carried out by adding state to this theory as an independent
dimension orthogonal to input. The simplest approximation to component behavior, instead of be-
ing a step function of the input, becomes a stepped plateau over these two dimensions. The changes
required in the theory are straightforward, but adding a state dimension significantly impacts the
performance of tools and makes it difficult to display revealing graphs. Initial experiments show
that subdomain refinement produces accurate system predictions and that the interaction between
state changes and input variation may be no more troublesome than complex input variation itself.

Adding state raises an interesting new aspect of component testing: portions of the two-
dimensional input–state space may be unreachable in principle. When a component is in place
in a system, part of the unreachability may result from the input profile it sees—this is no different
than a system failing to use part of the input space of a stateless component. However, part of the
state space may be unreachable no matter what inputs are received, which reflects the fact that state
is not just a hidden input parameter. Testing at the component level explores all regions, reach-
able and unreachable, because it samples states uniformly. In theories based on logic, unreachable
regions are those in which invariants are violated; it becomes a proof burden to show that they
are empty, that is, that the invariants are preserved. In the testing-based theory, the tools simply
display the behavior that would result if all states were reached.

The persistent-state theory and experiments have not been incorporated here because they
would make this beginning presentation hopelessly long.
9.4 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 very important, because it 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 the behavior of stateless components using subdomain testing, and uses these approximations to calculate approximate properties of systems built from the components.

Perhaps the most important insight to emerge from experiments with the theory is that subdomain boundaries are critical to accurate test-based specifications. When rapid behavior changes (including discontinuities) occur inside a subdomain, the predictions involving that subdomain are poor, and remain so even when the subdomain size shrinks. (To be sure, the overall prediction improves as subdomain size decreases, but only because the same error is confined to a smaller
and smaller part of the whole domain.) Component developers cannot know where crucial system subdomain boundaries will fall, but they can 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.

**Acknowledgments**

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 themselves values. This insight was crucial to the development of the theory presented here.

Zheng Tu, Milan Andric, Ben Buford, and John Christmann worked on the prototype tools and early versions of the experiments reported in Section 6.

I thank one anonymous referee for important suggestions on how to present these results.

**References**


