Tools and Experiments Supporting a Testing-based Theory of Component Composition

DICK HAMLET
Portland State University
PRE-PUBLICATION COPY — Please do not cite or quote

Development of software using off-the-shelf components seems to offer a chance for improving product quality and developer productivity. This paper reviews a foundational testing-based theory of component composition, describes tools that implement the theory, and presents experiments with functional and non-functional component/system properties that validate the theory and illuminate issues in component composition.

The context for this work is an ideal form of component-based software development (CBSD) supported by tools. 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 measured component descriptions and a system structure, a CAD tool synthesizes the system properties, predicting how the system will behave. The system is not built, nor are any test executions performed. Neither the component sources nor executables are needed by systems designers. From CAD calculations a designer can learn (approximately) anything that could be learned by testing an actual system implementation. The CAD tool is often more efficient than it would be to assemble and execute an actual system.

Using tools that support an ideal separation between component- and system development, experiments were conducted to investigate two related questions: (1) To what extent can unit (that is, component) testing replace system testing? (2) What properties of software and subdomains influence the quality of subdomain testing?

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and debugging

General Terms: Verification
Additional Key Words and Phrases: Experiments with composition of software components, synthesis of system properties, component-based software development (CBSD), CAD tool support for CBSD

1. INTRODUCTION

Component-based software development (CBSD) using off-the-shelf components is one approach to the problem of overwhelming complexity of modern software.

Author's address: Dick Hamlet, Department of Computer Science, Portland State University, P.O. Box 751, Portland, OR 97207, USA.
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 statements made in this paper.

Other engineering disciplines have been remarkably successful in defining and standardizing components from which large systems are designed and built. It is characteristic of successful component-based development that components are selected from a catalogue and the catalogue descriptions alone are required for systems design and evaluation. Systems design is done on paper, not by experimental construction. An important benefit of this ideal approach is that software can do part of the design work. So-called CAD (computer-aided design) tools help the system designer combine component catalogue descriptions into a system description.

The software version of component-based design is only in its infancy. There is disagreement about what constitutes a component, about the form of a catalogue description, and about the necessary or allowed forms of combination. The ideas and supporting tools available today are thought by some to be useful and important, but none comes close to the state of the art in (say) mechanical engineering CAD.

This paper describes tools and experiments the tools support, exploring a particularly simple and ideal theory of components and their combination based on testing. A component is taken to be an executable program. Its catalogue description is obtained by test measurements over a collection of input subdomains. The resulting description is an approximation to the component's functional and non-functional behaviors. Using the system-building constructs of sequence, conditional, and iteration, systems are designed from components. A CAD tool predicts system functional and non-functional behavior from the system structure and the component catalogue descriptions that comprise it. Only the component catalogue descriptions are required: the system designer needs no access to component code, not even binary executables.

By imposing restrictions on the components themselves, the theory allows a complete realization of ideal CBSD and the implementation of tools supporting experiments that expose issues and problems in its use. The restrictions are:

—Component input and output are single floating-point value domains.

—Components may have local persistent state, but only from a single floating-point value domain.

—There is no concurrency.

These are severe restrictions, but they are consistent with the bulk of existing testing theory that began with the work of Goodenough and Gerhart [Goodenough and Gerhart 1975]. They allow an ideal separation between component development and system design, and if they are observed, powerful tools can be implemented to support CBSD. Without the restrictions, it is problematic what tools can accomplish. For example, detailed practical models like UML are supported only by syntax-checking and bookkeeping tools; nothing remotely like the automatic synthesis of our CAD tools is imaginable.

In what follows we speak as if our theory and supporting tools were being used by practicing software developers, but this is only a presentation device. Before these ideas can be used in practice, many difficult technical problems would have to be solved. In the restricted setting described in this paper, our purpose is to gain understanding of the way in which components might be used ideally, the problems that arise, the way in which tools can help solve those problems, and limitations of tool support.

The remainder of this paper is organized as follows: Section 2 is a brief summary of theoretical background. Section 3 describes implementation of supporting tools with expository examples of their use. Section 4 describes experiments exploring and validating the theory. Section 5 collects some applications of the theory supported by the tools. Section 6 describes planned extensions of this project.

2. THEORETICAL UNDERPINNINGS

A major deficiency of software-testing theory is that it is not "compositional." Independent test results from software elements C_1 and C_2 cannot be directly combined because interacting test outputs and inputs do not match. For example, if C_1 has been tested on input X with output Y and C_1 and C_2 are placed in sequence, it is unlikely that C_2 will have been tested on input Y, so the composite output on input X cannot be predicted.

Subdomain testing [Howden 1976] provides an approximation that solves the problem of mismatched output/input; our theory of software component composition is based on subdomains.

2.1 Stateless Components

Because the behavior of programs that retain state is potentially much more complicated than without state, stateless theory is given first. It exhibits many characteristics of the more general case.

Consider again C_1 and C_2 in sequence, but now with independent subdomain test data. There is a decomposition of each input space into a finite disjoint set of subdomains, and corresponding subdomain-output values obtained by testing. If subdomains are relatively homogeneous, a single output value approximates the behavior on each subdomain. Given any input X, it falls in some C_1 subdomain. Suppose inputs in that subdomain have average output Y. Y falls in some C_2 subdomain where the average output is Z. Then Z approximates the output of the sequence $C_1; C_2$ on input X and thus independent component test results *can* be combined to get (approximate) system results. The calculation is not accurate unless the subdomains are perfectly homogeneous so that all outputs in each subdomain are the same (and hence equal to the average there). But one might expect that by shrinking the subdomain size the accuracy will improve.

For numerical input domains, there is a more accurate approximation to the component behavior. By fitting a straight line to test data within a subdomain, output variations over the subdomain are captured. Consider again $C_1; C_2$ where the behavior on each of their subdomains has a linear approximation. Given input X, it falls in some C_1 subdomain on which the line is (say) $\lambda x(mx + b)$. Then the approximate output is $Y = mX + b$. Y falls in some C_2 subdomain with line $\lambda x(m'x + b')$. Then the composite output is $Z = m'Y + b' = m'(mX + b) + b' = mm'X + m'b + b'$. It is important that the composite result is also a line (of slope mm' and intercept $m'b + b'$). The linear approximation is the highest order with this closure property.

The subdomain approximation with constant output on each subdomain will be called the *step-function* approximation; the one with linear behavior will be called the *piecewise-linear* approximation. Subdomain test results for any program (component or system) consist of:

(1) A finite list of disjoint subdomains whose union is the program input domain;

(2) An output value for each subdomain. For the piecewise-linear approximation this value is a (slope, intercept) pair for the line.

Subdomain test results are thus an acceptable form for a component 'catalogue entry' as described in Section 1. It can be measured for a given component, and as we see next, it can be calculated for systems.

The heart of a testing theory of component composition based on subdomains is that from independent subdomain tests of two components, the results of a subdomain test of those components in sequence can be *calculated*, without ever forming the series system or testing it. There is an algorithm for calculating an approximate "equivalent component" for a sequence.

Algorithm B (stateless functional sequence composition, step-function approximation).

Input: Subdomain test results for each of two components C_1 and C_2.

Output: Subdomain results for the series system $U = C_1; C_2$.

Algorithm: The subdomains of U are exactly those of C_1. Let S be an arbitrary subdomain of U, and Y the C_1 output there (from the C_1 test results since the subdomains of U are those of C_1). If Y falls in some C_2 subdomain with output Z (from the C_2 test results), then the output of U on S is Z; otherwise, U's output for S is undefined.

A similar construction for the piecewise-linear approximation is somewhat more complicated, because the subdomains for U may be fragments of those of C_1. Details are given in [Hamlet 2007b].

"Glue code" is a name given to fragments (often of a surrounding control program) needed to adjust the interfaces of components that do not quite fit together. In the restricted context of this paper, there is no surrounding program, yet something like glue code is often needed, for example to scale an output range to match a following input domain. It is always possible to insert a series component to make the match.

To synthesize an equivalent component[1] for:

$$\texttt{if } T \texttt{ then } B \texttt{ else } C \texttt{ fi},$$

intersect the subdomains on which T is *true* with those of B, and on these copy the values from B's test results; intersect the *false* subdomains of T with those of C, and here copy C's test results. The intersections create refined system subdomains. If there is no `else`, for C use an identity component with zero run time, which can be perfectly captured in the piecewise-linear approximation.

It is something of a surprise that one can algorithmically find an equivalent component for a loop from its component descriptions. The approximate nature

[1] In the conditional and the loop to follow, the usual flowchart convention is that the input to T is also provided to the following component(s). T's output is used only to select the path.

of the subdomain test data enters essentially in the construction of an equivalent iterated component. If the loop is:

`while` T `do` B `od`,

it can be unrolled any finite number of times:

`if` T `then` B `fi`; `if` T `then` B `fi`;...; `while` T `do` B `od`.

In the step-function approximation, suppose that the equivalent component for `if` T `then` B `fi` has K subdomains. Then in at most K unrollings of the loop the iteration will either be seen to be non-terminating or the composite subdomain values will be obtained. Intuitively, each unrolling may remove one or more subdomains from consideration (because T goes *false* on them), in which case the synthesis algorithm requires composing in series at most K copies of `if` T `then` B `fi` with no residual loop. Or, no subdomains are removed on some unrolling, which signals non-termination. The equivalence of a loop with its unrolling applies only to output, not to run time. A modification to the equivalent component for `if` T `then` B `fi` is required in the repeated series composition to avoid repeating T's run time incorrectly. (For the piecewise-linear approximation the bound on number of unrollings is more complicated.)

Non-functional properties of components are sometimes compositional in the same way. In this paper we mostly use run time as the example. Thus add to Algorithm B:

> On C_1 subdomain S let the step-function approximate run time of C_1 be R_1. For the subdomain of C_2 in which C_1's S output falls, let the approximate run time of C_2 be R_2. Then the approximate run time of series system U on S is $R_1 + R_2$.

The three system-building constructs of sequence, conditional, and iteration are sufficient to construct an arbitrary software system [Boehm and Jacopini 1966]. Straightforward algorithms like Algorithm B construct, in both the step-function and piecewise-linear approximations, an equivalent component for sequence, conditional, and iteration constructs. Details are given in [Hamlet 2007b].

2.2 Components with Persistent Local State

In the ideal CBSD paradigm state must be encapsulated within components; if there were a global state, some extra, 'system-level' code would be required to manipulate it. Each component is allowed a single persistent floating-point state value, kept in a permanent disk file with a unique name. A step-function approximate description of a component involves two orthogonal sets of subdomains, one for input, the other for state. The description itself (catalogue entry) comprises three two-dimensional tables of values: as in the stateless case, one for output and one for run time, and an additional table for output states. Thus the graphs of a catalogue-description approximation, instead of being step functions over the input, are three step-plateau functions, rectangular flat steps above the rectangular subdomains in the input–state plane. To measure the average values of these plateau functions requires only sampling in each (input×state) subdomain. Systematic sampling is not really correct; sampling will be discussed in Section 3.1 to follow. For the moment, imagine that step-plateau descriptions are available for a collection of components.

The basic algorithm for calculating from two such descriptions a description of their series system is a straightforward adaptation of Algorithm B. The composite system state is a cross product of local states.

Algorithm B′ (functional sequence composition with state).

Input: Subdomain test results (three step-plateau functions) for each of two components C_a and C_b.

Output: Subdomain step-plateau functions for the series system $U = C_a; C_b$.

Algorithm: Let C_a have a typical input subdomain J_a and typical state subdomain H_a, while C_b has typical state subdomain H_b. The input subdomains of U are the same as those of C_a, while U's state subdomains are like $H_a \times H_b$. Thus a typical subdomain of U is $S = J_a \times H_a \times H_b$. Let Y be the C_a output on $J_a \times H_a$ (from the output step-plateau function of its test data). If Y falls in some C_b input subdomain J_b, and the C_b output for $J_b \times H_b$ is Z (from the C_b test data), then the output of U on S is Z; otherwise, U's output for S is undefined.

Similarly, the run-time and result-state values for S are obtained from the other two step-plateau functions of the components.

In the stateless case, Algorithm B yields tables for the system that have exactly the same form as its input tables. Algorithm B′ is not as tidy: its subdomains are like $J_a \times H_a \times H_b$, one dimension higher than the input tables. Its output states are not the single-value ranges of input states, but pairs. If one of the two series components is stateless, the system state is identical to that of the other component and the dimension increase does not occur.

The adaptation of the stateless algorithms for calculating an equivalent component for the conditional and iteration constructions are also straightforward, but there are a number of complications. In a conditional, the result has a potential for a triple cross-product state, one dimension from each of the three components involved. In the loop the state of the body component does not multiply (although the result may have a two-dimensional state if both the guard and the body have state), but unrolling is less useful, since it applies only to output, not to either run time or state.

2.3 Proving the Theory

Algorithm B′ and the others for calculating the results of combining catalogue descriptions for components are almost entirely a matter of complicated bookkeeping, pulling values from the component subdomains and combining them to populate the system subdomains. It is usual in mathematics to argue for the correctness of such algorithms as "obvious," or "by construction." What this means is that once the underlying ideas are understood, anyone would agree that the algorithms can be made to work. There may very well be blunders in the bookkeeping, but if a mistake comes to light, it will be evident how to adjust the construction to fix it. Sometimes a more formal proof is given for such algorithms, usually by induction on the size of the instance (here perhaps the number of subdomains and the number of system constructs). But everyone agrees that such proofs are "uninformative," that is, they are useless in understanding the algorithms.

In the present case, these algorithms are the basis of CAD tools that implement synthesis of arbitrary systems. Since the algorithms are straightforward, so is the coding of these tools, but blunders are even more likely. Algorithm B′, for example, involves more than ten pages of Perl code. Fortunately, there is an unusual way to validate the tools, described in Section 3.3.

3. TOOL SUPPORT

Research-prototype tools were implemented in Perl to:

(1) Test and approximately describe components using subdomain testing (that is, to create catalogue entries for components), and

(2) Make the theoretical calculations described in Section 2 to synthesize properties of a system built from components (that is, to calculate an approximate equivalent component for a system).

The second set of algorithms is the core of a 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 7000 lines of non-comment Perl code, but there is some overlap between the stateless and with-state versions. The CAD tool that handles both cases is about 1800 lines.

3.1 Component-testing Catalogue-entry Tools

A component developer has executable code, and must make measurements to create a catalogue entry, that is, a list of subdomains and an approximation to functional-, run-time, and result-state values on each subdomain. Our component-testing tools use a configuration file containing the subdomain list, sampling frequency, and name of the component executable file. From this information a catalogue entry is created by attaching to each subdomain a triple of values (output, run time, state). In the step-function approximation, values are averages obtained by sampling over the subdomain. The component-testing tool produces a plot of the approximation obtained by sampling compared to the measured component behaviors, with the measured relative root-mean-square error for each subdomain.

3.1.1 *Stateless Catalogue Descriptions.* Figure 1 shows the step-function approximation to a stateless artificial component C_0 constructed to illustrate the tools. The horizontal axis in Fig. 1 is marked with the subdomain boundaries (light tics crossing the axis), chosen by subdividing an input domain of [0,10) into 25 intervals, smaller where the function changes more rapidly. To obtain the average values each subdomain was sampled three times. The approximation is perfect only for functional values on the interval [4, 5) where the component's actual output is constant. The weighted average r-m-s error is 2.8% for the output and 2.5% for the run time.

For the stateless case[2] the tools also implement a more accurate approximation by fitting a line to the measurements in each subdomain. For C_0 this piecewise-linear approximation is perfect, because the component behaviors *are* linear. With

[2] It was decided not to implement piecewise-linear approximation for components with state; the rationale is given in Section 6.1.

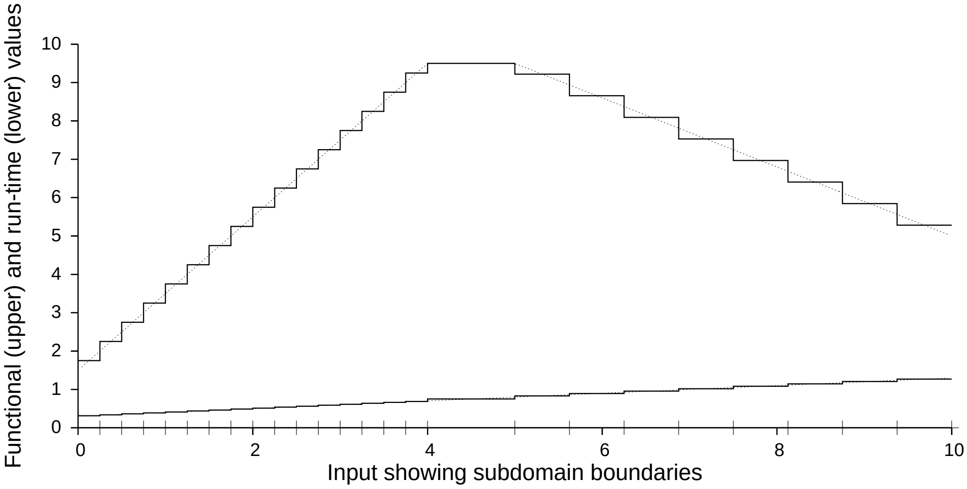


Fig. 1. Approximated behavior (solid line, for the step-function approximation) of a stateless component C_0 and its actual behavior (dotted line)

this better approximation, the tool output shows measured and predicted curves for C_0 that coincide, and zero r-m-s error in every subdomain.

3.1.2 *Catalogue Descriptions for Components with State.* As an illustrative component with state, choose C_m that uses its state to define four 'modes.' Figure 2 displays the output step-plateau graph[3] for C_m. Inputs in [0,1) are used to set the mode (state) value; this behavior comprises mode 0. In the other modes the component has one of: (mode 1) linearly increasing output; (mode 2) behavior similar to C_0 of Section 3.1.1; (mode 3) parabolic output. Defining and storing modes that subsequently influence behavior is a very common use of local state, used notably by interactive applications whose users set 'preferences' (the modes). The input domain for C_m was chosen to be [0,10) and the state domain [0,5). There were 20 input subdomains and 10 state subdomains. Each of the 200 cross-product subdomains was systematically sampled nine times.

The systematic sampling used to obtain Fig. 2 is natural, and corresponds to the way states are often sampled in testing practice. However, systematic state sampling is wrong in principle. Unlike input, state is not an independent dimension under the control of the tester. What really happens is that a component initializes its local state value, then subsequent inputs drive it from state value to state value completely under program control. Each such sequence of inputs starting from state initialization is repeatable, but within a sequence a repeated input may lead to different results because of the changing state. Therefore, only certain states are possible. Systematic sampling may produce phony results, by setting an "input state" that never actually occurs. By changing the sampling to sequences of random inputs, we can see what C_m really does (Fig. 3). The four modes are clearly visible in Fig. 3, and other states between and beyond them are seen to be infeasible. Infeasible states do not appear in executions, so neither do they appear in measured

[3] For this example run time is omitted from the discussion.

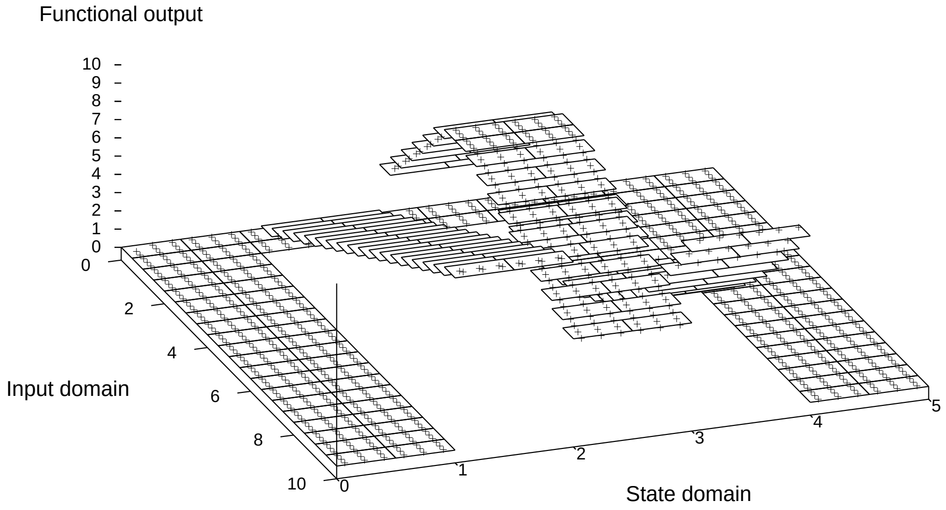


Fig. 2. Approximate functional behavior (rectangles) of component C_m and its actual behavior (crosses) sampled systematically

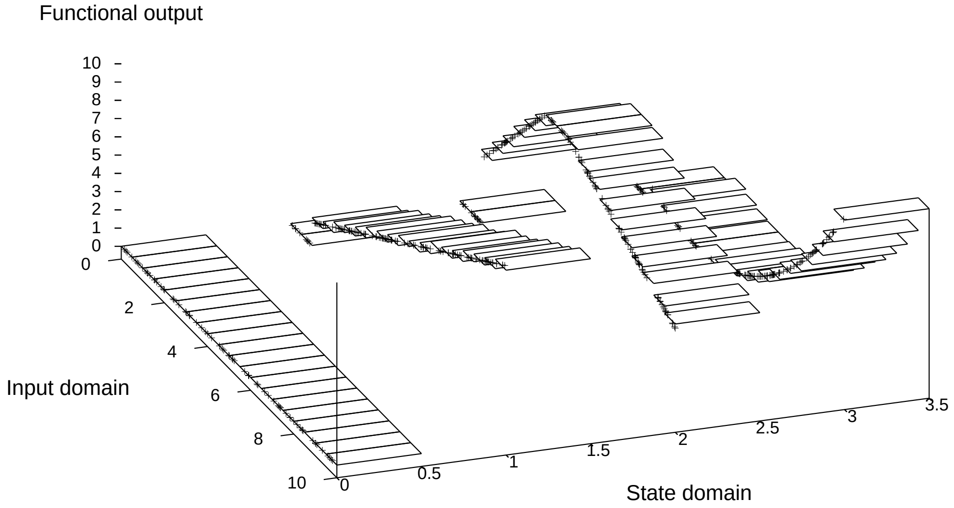


Fig. 3. Approximate functional behavior (rectangles) of component C_m and its actual behavior (crosses) sampled with input sequences. Compare Fig. 2 in (say) subdomain [9,9.5)×[.5,1)

approximations. Figure 3 was obtained by testing C_s with 30 sequences of uniform random inputs, sequence lengths uniformly distributed in [0,30]. In these sequences there were a total of 474 input values. The weighted r-m-s error in Fig. 3 is 2.3%.

A component with local state has result-state behavior as well as output: when in a given state, an input drives it to a new state. Figure 4 shows result-state behavior for C_m, sampled correctly with the same random input sequences used for Fig. 3. In this case the r-m-s state error is 0.3%.

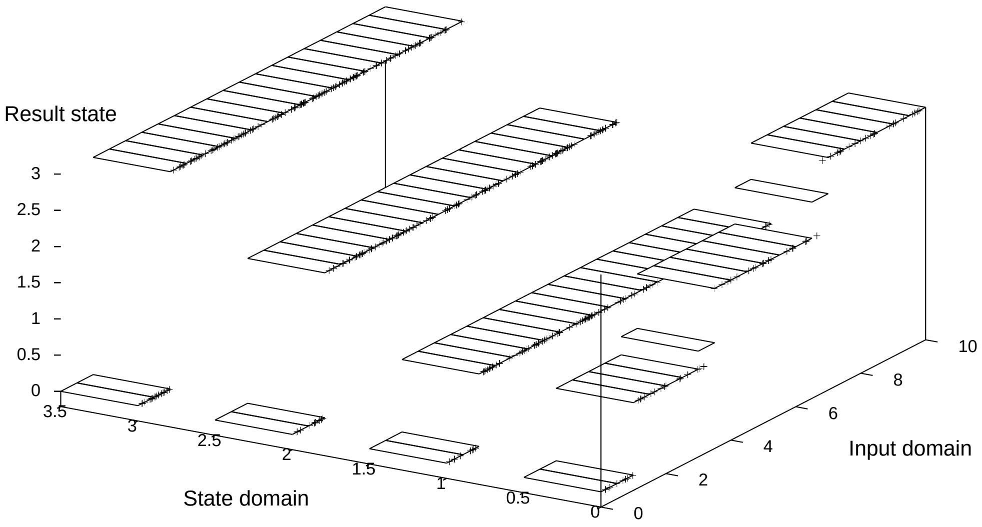


Fig. 4. Approximate result-state behavior (rectangles) of component C_m and its actual state behavior (crosses) sampled with input sequences

The difference between testing with systematic state sampling and the correct input-sequence sampling need be no worse than in the C_m example. Systematic sampling gets the right answer for feasible states but may include infeasible states. However, one style of coding can lead to completely erroneous output and run-time behavior for systematic state samples. If a component processes a state "reset" condition by taking no action except to create a base state with a throw-away output like a 'Done' message, systematic state sampling can show output 'Done' and its run time as the only possibilities. The code intends that behavior occurs in a sequence of executions, the first to create state and subsequent ones to use that new state; but none of these subsequent executions occurs in systematic state sampling. Unless care is taken, each of the systematic samples appears to be the "reset."

3.1.3 *Summary of Component-description Tools.* A component developer can use these tools to examine the behaviors of a component being implemented. The developer is working from a specification, which serves as a test oracle for the code, usually applied by hand. The measured data of graphs like Fig. 1 can be compared to the oracle to verify that the implementation has not failed. Other verification methods will probably be applied as well. For example, it is common to define a collection of specification-based functional subdomains for testing, or to use uniform-distribution random testing, or informal proving techniques. These other verification techniques are unrelated to our tools. In particular, the subdomains we use are not likely to be ones that come from specification-based tests.

Once the developer is confident that the component code is behaving correctly, our tools create a catalogue description, which is their primary purpose. It is tempting to call that description the component "specification," following standard usage in the world of mechanical and electrical components. But in this paper we avoid "specification" with any meaning other than an a priori list of requirements that should have been implemented. The catalogue description is an approximation to the implemented component's actual behavior. That may be the desired behavior, but not necessarily, unless the developer was both careful and lucky in verification.

The catalogue entry itself is a table defining (for example) the step-function shown in Fig. 1 or the step-plateau functions of Figs. 3 and 4. The description can be 'executed' by table look-up as described in Section 3.3; such an 'execution' of an approximation would reproduce the step/plateau functions shown in the figures.

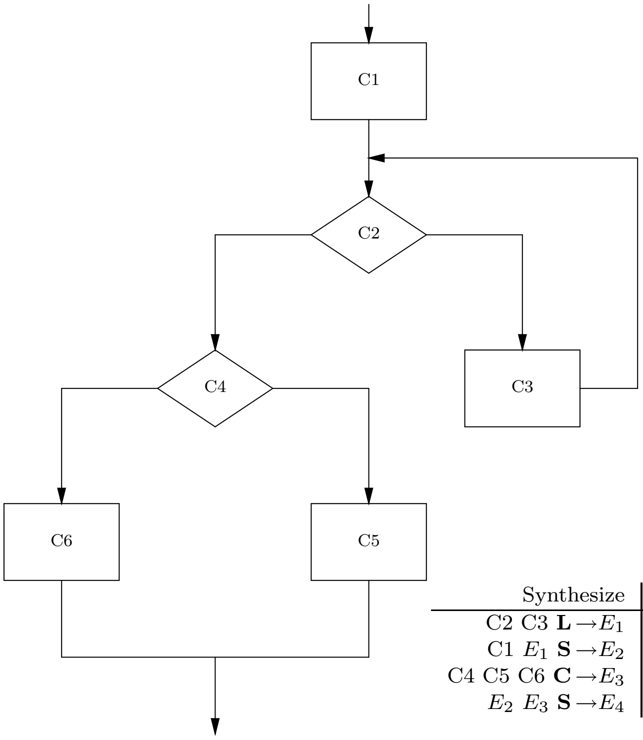
3.2 System-design (CAD) Tools

A systems designer designing a component-based system works from a components catalogue and tentative ideas about how these components should be combined. To explore this process, we have implemented a CAD tool that takes as input a collection of component approximate descriptions (that is, catalogue-entry tables produced by the specification tools of Section 3.1) and a description of the system into which they are to be combined. Its output is a system prediction in the same form as the component descriptions; that is, an 'equivalent component approximation' for the system. The system form to be used is given to the tool as a reverse-Polish string of sequence, conditional, and iteration operators whose operands are components. The CAD tool uses the algorithms of Section 2 to synthesize each construct as a subsystem in the form of an equivalent component, then combines these in turn, ultimately synthesizing the complete system as a single table that can be executed by table lookup.

A support tool produces a plot of the predicted behavior. The predictions can be compared with values obtained by running a system formed by linking together the actual component code in the given structure.

To illustrate the CAD tool, consider the simple system structure shown in Fig. 5, which was chosen to illustrate all three constructs. Fig. 5 also shows the reverse-Polish description with operators **<u>S</u>**eries, **<u>C</u>**onditional, and **<u>L</u>**oop, and the successive synthesis steps to produce equivalent components, culminating in E_4 for the system.

3.2.1 *An Example using Stateless Components.* To keep the exposition as simple as possible at first, only two stateless components are placed in the six boxes labeled C1 – C6 in Fig. 5. An input domain of [0, 10) is arbitrarily chosen for the example. The conditional-test components C2 and C4 are written as the same component T that returns *true* only in the interval [2,6), with a constant run time of 0.2.



Synthesize	C1 C2 C3 **L** **S** C4 C5 C6 **C** **S**
C2 C3 **L** →E_1	C1 E_1 **S** C4 C5 C6 **C** **S**
C1 E_1 **S** →E_2	E_2 C4 C5 C6 **C** **S**
C4 C5 C6 **C** →E_3	E_2 E_3 **S**
E_2 E_3 **S** →E_4	E_4

Fig. 5. A simple system structure and its synthesis steps

Components C1, C3, C5, and C6 are copies[4] of C_0, whose behavior is shown in Fig. 1. The step-function approximation is used. First the equivalent component E_1 is synthesized for the loop of C2 and C3. Three unrollings are required for termination in 67 subdomains. Next the series combination of C1 with E_1 is synthesized to E_2. The conditional equivalent component E_3 is synthesized from C4, C5, and C6. Finally, E_2 in series with E_3 produces a system equivalent component E_4.

Figure 6 shows the resulting predictions with the actual system behavior systematically sampled 300 times for comparison. The first thing to notice about the actual system behavior is that even for this simple expository example it is complicated. There are four discontinuities that result from the conditionals, but their locations are not intuitively obvious. (The approximation fails to predict the second discontinuity.) The approximation errors are largest in subdomains [0.25, 0.50) (second from left in Fig. 6) and [8.75, 9.38) (second from right), where system discontinuities do not fall on a subdomain boundary. The weighted average r-m-s error is 4.9% for output and 3.3% for run time. A detailed investigation of system predictions for stateless systems, exploring the way in which the accuracy depends

[4] Replication of a single component in a system may reuse its unique catalogue entry; the synthesis algorithms make no use of any names in the component code. In executing the actual system for validation, it is also OK to reuse the single copy of a component's code, but only in the stateless case. When there is state, it is implemented with a named file, and that name must be changed in replications of a component.

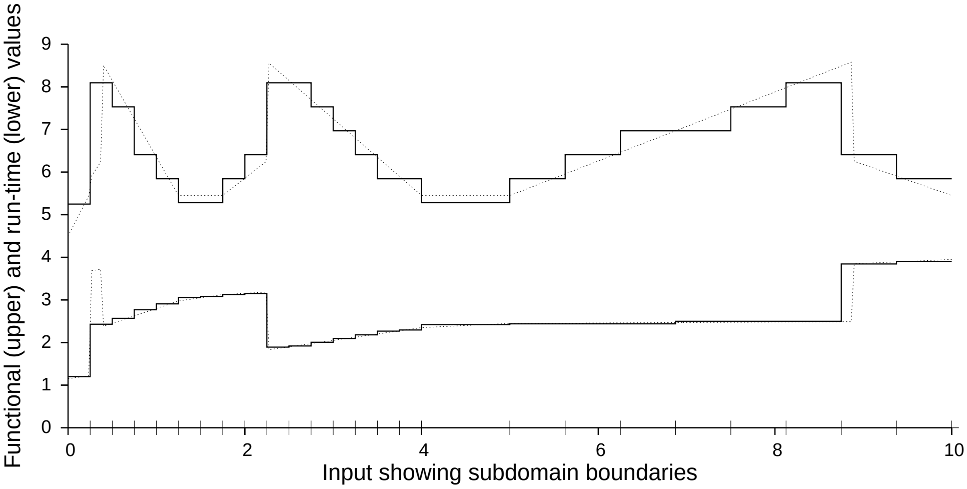


Fig. 6. Predicted (solid line) and actual (dotted line) behaviors for a simple system

on the accuracy of component approximations, is presented in Section 4.1.

3.2.2 *An Example using Components with State.* It is more difficult to give an expository example of system synthesis using components with state, because of the way in which system state grows dimensionally. Unless a system has all stateless components but one, its behaviors will depend on more than two dimensions so they cannot be easily graphed for visualization. We therefore choose for illustration the simple system of Fig. 5 containing just one component with state: all the non-conditional components are C_0 except C1 is taken to be C_m from Section 3.1.2. Figure 7 displays the results of the CAD calculation predicting system functional behavior. The four modes of C_m are visible, and mode 2 (third from left, second from right in Fig. 7) reproduces the behavior of the stateless system of Fig. 6, because C_m in mode 2 acts like C_0. The measurements (plotted crosses) in Fig. 7 were obtained with 50 random sequences of inputs containing 1212 points, as for C_m in Fig. 3. The weighted average r-m-s error is 5.4%. Since only C1 has state, the system state is a copy of it, and the system state prediction graph is the same as Fig. 4. Further discussion of system-synthesis accuracy with state will be deferred to Section 4.2.

3.3 'Executing' Component Catalogue Entries

Component catalogue entries themselves can be 'executed' by table lookup. That is, to obtain the approximate behaviors of a component for input x and state s, look up (x, s) in the subdomains of its catalogue description and return the functional-output, run-time, and result-state values stored there for the subdomain containing (x, s). In the step-function approximation the values returned are constant for all (x, s) in the same subdomain. The 'equivalent components' synthesized by the CAD tools are in the same form, and they can be similarly 'executed.' This is the sense in which a calculated system is available to its designer 'on paper.' Anything that could be learned from executing the actual assembled system can be learned (approximately) by table lookup in the calculated equivalent component for the

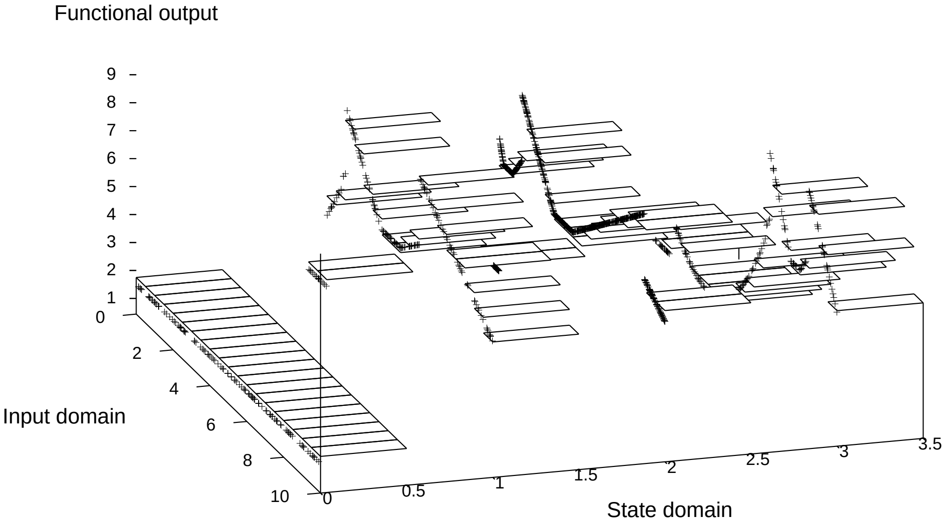


Fig. 7. Predicted (rectangles) and actual (crosses) output for a simple system

system.

It is sometimes difficult to conceptualize execution using a description that does not execute its parts. The computed equivalent components are static, and do not go through the motions that one intuitively expects. For example, the equivalent for a conditional does not, when executed, test its condition. A loop equivalent does not in any sense iterate. If the theory involved were a formalism like Floyd-Hoare logic [Floyd 1967] or Mills functional calculus [Mills et al. 1987], there would seem nothing strange about composing descriptions to get a new description. But because ours is a testing theory, one falls into the idea of execution when there is none. However, there is an intermediate way to look at the theory which helps to bridge the intuitive gap. Given the system structure, executable catalogue entries can be treated as code by surrounding each with a table-lookup wrapper. Then an approximate system can be assembled by linking these wrapped tables together in the given structure. Such a system, when executed, has exactly the same behavior[5] as the CAD-calculated system equivalent, but it *does* go through the expected operations: the conditional components do test inputs (by table-lookup), the loops do iterate (the table lookups), and so on.

This intermediate system formed from the approximate component catalogue descriptions has two very useful applications: First, it serves as an oracle for checking the correctness of the CAD tools. If CAD validation tests such as described in Section 3.2 are run not against the real system code, but against the table-lookup version, they should show zero errors everywhere. The correctness of the lookup

[5] There is an exception in the case of a conditional involving more than one component with state, described in Section 4.2.1 and in Section 6.1.

system depends only on proper measurement of each component description and proper linking for each of the system constructs. Those implementations are simple and can be seen to be correct early on. They are then used in every case study to certify that the CAD calculations (by no means simple or obviously correct) are right. Second, when it is difficult to derive information that intuitively depends on execution from the CAD approximation, the table-lookup system can be used instead. For example, the trace facility mentioned in Section 5.2 below was implemented only for executions. But traces for CAD approximations can be obtained with the same tool, by using the table-lookup components.

Section 3.5 discusses the performance of the tools, comparing the times required to execute a real system, to execute a system by table lookup in each of its component catalogue entries, to make the CAD calculation, and to execute the calculated system equivalent.

3.4 Artificial Components Constructed to Order

The components employed in the expository examples of this paper are artificial. We began with 'real' components, but found that they were not good at stressing the theory and tools or for gaining insights about the CBSD process. What appeared at first an expedient turned out to be a cornerstone of this study.

Initially, we tried to create or find 'real' components whose behaviors would make good case studies. 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: The predictions were perfect. However, closer examination showed that this was a trivial consequence of the simplicity of the example: the component behaviors were constant on obvious subdomains so the example was no real test of the theory. Next, we crafted a Java component that calculated by brute-force trial the spacing between the first two primes larger than its input, and studied this component in series constructions. The theory run-time predictions were wildly inaccurate, which we traced to measurement errors when the Java run-time environment did an unexpected garbage collection; we also had difficulty with repeatability of the 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 [Hamlet et al. 2003], but detailed study of the example showed that despite its complications it was a trivial application of the theory.

Because the prime-spacing example was long-running, 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 `stderr`. In retrospect it is an obvious insight to realize that waiting for actual execution is not necessary. If a phony component is created that simply writes run-time values to `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 the easy creation of arbitrary component run-time functions.

(2) It 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 functions can be made arbitrarily complex in a direct way, without the need to consider what a component is 'really' doing (or to wait for it to do it).

Closed-form formulas were used in phony components to produce interesting behavior functions. For example, the following is the Perl code for the component C_0 whose behavior is plotted in Fig. 1:

```
$x = <STDIN>;
if ($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 "$y\n";
print STDERR "$t\n";
```

A generator was also written to create phony stateless components behaving like 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 prime-spacing component, where we were able to closely reproduce its behavior, including the discontinuities resulting from the Java garbage collection.

Testing methods have traditionally been validated using a small collection of 'toy' programs[6]. The constructed components employed here might be thought worse than 'toys.' However, our example components are often harder on the theory than real ones would be: they are contrived precisely to stress it as real components failed to do in our initial attempts. The component C_0 used in the exposition of Section 3.2.1, for example, has input-output behavior that varies widely and non-monotonically over its domain. Although the examples of Section 3.2 are contrived to demonstrate the tools supporting our model, they were not adjusted to exhibit the features that they do. Similar features appear in every example in which component behaviors have substantial variation, as discussed in Section 4.1.

3.5 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 work once, to be used by all subsequent systems designers. The performance of CAD

[6] The triangle-classification program first employed by Glenford Myers [Myers 1979] 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.

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

A companion paper [Hamlet 2007b] analyzes the performance of the stateless synthesis algorithms in detail, but for present purposes it is enough to note that CAD performance depends only on the number of system constructs (roughly the number of components in a system) and the number of subdomains in the components being combined. On the other hand, actual system execution time depends on the number of test inputs, the number of components, each one's actual run time, and the iteration counts for loops. When 'executing' a table-lookup system built from the components' catalogue entries as described in Section 3.3, the run time for any input to any component is a small constant but the subdomain must be looked up, requiring a time that depends on the number of subdomains. Only one input per subdomain need be tried in the step-function approximation. These parameters are largely distinct for the three ways of getting information about a system, so time comparisons can be contrived to favor any of the three schemes. But when the number of actual test inputs and actual run times are large relative to subdomain counts, the CAD tools have a large advantage, the table-lookup system executions are second best, and actual system execution is slowest.

On a 1.7 Ghz PC the synthesis time for the example stateless system of Section 3.2, which has 6 components, 5 synthesis steps, and 25 subdomains, is about .88 s. The table-lookup execution time is about 2.9 s and for actual execution 23 s. Since the components are artificial, the measured actual run time is nominal, reflecting only the overhead of running a process. If the actual average system run time were 100 ms and 20 samples were taken per subdomain, the system would take an additional 50 s to test. The large disparity for small examples reflects mostly a difference in overhead: the calculations do table manipulation while the executions require process initiation. If the system run time or the number of system test samples increases, the actual test time increases while the CAD-synthesis time does not change[7].

When components keep persistent state, the sampling domains increase in size quadratically, which increases the disparity. Furthermore, executions now have to manipulate the files that hold state values, another high-overhead operation.

4. VALIDATION EXPERIMENTS

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. Continuity allows the experimenter to interpolate with confidence between relatively sparse samples. On the contrary, software is a human creation that can be changed at will and is intrinsically discontinuous, so that sampling its behavior (that is, testing it) is

[7] The argument is biased in favor of the theory because the reason for making more actual test runs would be to improve the quality of testing. To be fair, more subdomains should then be used in the theoretical cases. The table-lookup times increase only when there are loops that take more iterations to complete.

often misleading. 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. Perhaps it is better to call theory a 'model' of real software. 'Validation,' checking the correspondence between model and reality, is not as meaningful for software as for physics. A good model simplifies, and thereby distorts reality, but although this necessarily makes it 'invalid,' its explanations can be useful. We do not need to experiment with an abstraction to learn about software—we can experiment with the thing itself. But when experiments with real software are incomprehensible or inconclusive, it may be better to work with a model.

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 model, 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[8].

Test the implementation of tools. For simple cases it is possible to calculate the theoretical results by hand, and cases of perfect approximation should produce perfect predictions. There is a unique opportunity for checking our tools because the predictions of the theory can be obtained in two entirely different ways and compared, as described at the end of Section 3.3.

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.

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 software and its test subdomains influence the prediction accuracy?

[8] The use of a 'transfer matrix' in our initial theory [Hamlet et al. 2001] was such a definitional mistake. We also failed initially to understand that loop synthesis was deterministic and that the unrolling of loops did not apply to run time and state. These mistakes were immediately discovered by experiments.

4.1 A Systematic Subdomain-refinement Experiment (Stateless Components)

A number of system structures were investigated using constructed stateless components whose behaviors vary 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.

A simple but typical case study uses the structure of Fig. 5, but with the differing component behaviors listed in Table I. An input domain of [0,10) was arbitrarily selected for the case study. Component C1 is C_0 shown in Fig. 1. Figure 8 shows a measured description of another component (C6 in Table I) using the piecewise-linear approximation with 20 subdomains. For component C6 (as for all these

Component	Function $y = f(x)$	Run time $T = g(x)$
C1	$y = \begin{cases} 2x + 1.5 & x < 4 \\ 9.5 & 4 \le x < 5 \\ -.9(x-5) + 9.5 & x \ge 5 \end{cases}$	$T = .1x + .3$
C2	$y = \mathit{true}$ iff $3 \le x < 5$	$T = .2$
C3	$y = .2(x-6)^2 + 1$	$T = .1$
C4	$y = \mathit{true}$ iff $(1 \le x < 2) \vee (5 \le x < 6) \vee (x \ge 8)$	$T = .3$
C5	$y = \begin{cases} 2\sin x + 5.2 & x < 5 \\ 3\cos^2 x + 4 & x \ge 5 \end{cases}$	$T = .6 - .04x$
C6	$y = 7\lvert\cos(x^2/8)\rvert \mathrm{e}^{-x/7} + x/2$	$T = x/12 + .1$

Table I. Component behaviors in the system of Fig. 5

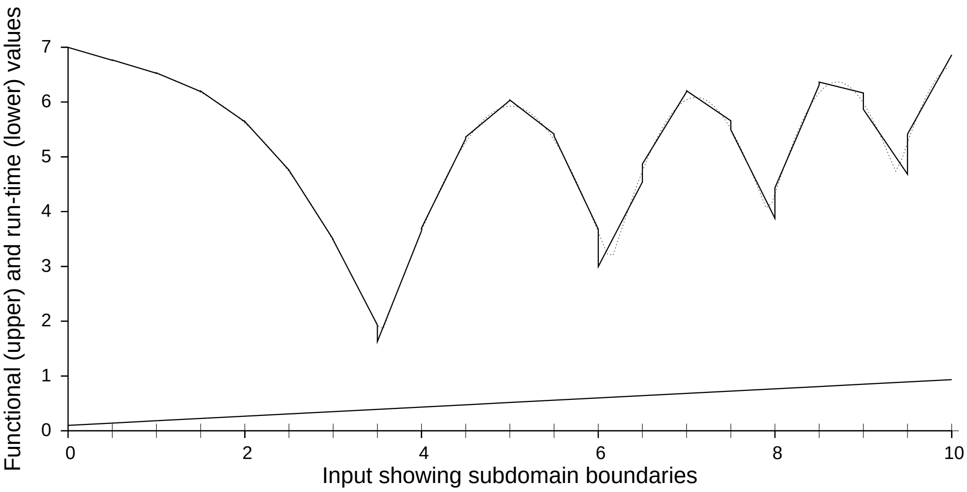


Fig. 8. Measured (dotted line) and piecewise-linear approximate (solid line) behavior of component C6

components) the piecewise-linear run-time approximation is perfect; Fig. 8 shows that the piecewise-linear functional approximation to C6 is quite accurate except in a few subdomains, e.g., just above input 6 where a cusp falls inside [6,6.5).

The example is explicit and concrete: every detail is given in Table I and Fig. 5. The corresponding measured system behaviors are shown in Fig. 9.

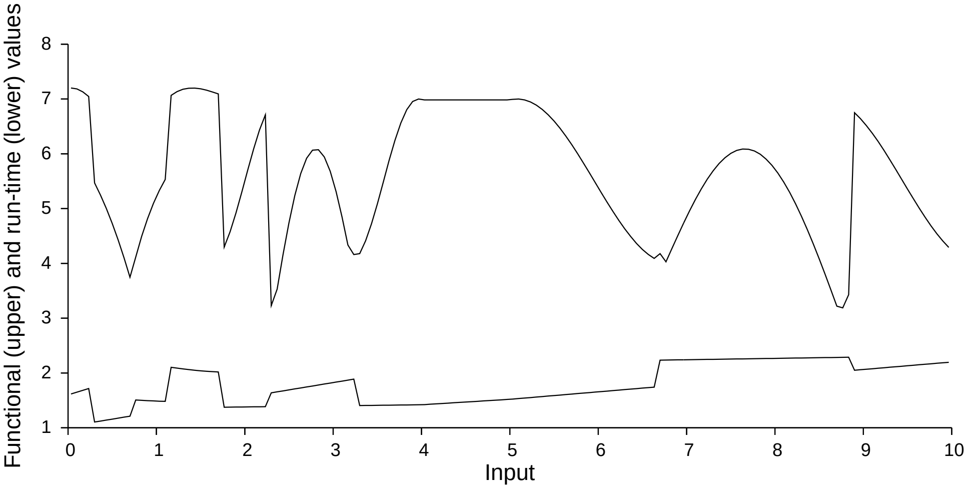


Fig. 9. Measured behavior of the case-study system

Table II summarizes experiments on component behavior using the step-function approximation as subdomains are refined. In the table, accuracy measures that

System	Component									
sub-	measurements		System predictions and measurements							
domain	rms % error		Functional % error				Runtime % error			
count	Function	Runtime	Overall	Max	Mean	> 5	Overall	Max	Mean	> 5
4	12.88	6.95	20.93	37.15	19.26	4	14.12	13.60	7.40	2
8	7.48	3.55	17.36	17.81	10.78	7	9.29	14.30	4.97	4
16	4.13	1.77	15.32	22.89	7.93	7	7.75	21.97	6.26	7
32	2.07	0.90	7.47	16.19	3.77	9	1.69	15.71	1.14	2
64	1.05	0.45	4.29	29.78	1.97	3	0.91	7.22	0.44	2
128	0.52	0.22	2.01	4.41	0.88	0	0.43	7.78	0.20	1
256	0.27	0.10	1.15	16.40	0.53	1	0.18	7.52	0.12	2
512	0.15	0.05	0.73	28.78	0.29	2	0.13	18.66	0.09	3
1024	0.07	0.05	0.21	3.79	0.12	0	0.03	7.19	0.03	1

KEY: "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 "Runtime % 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 II. Prediction accuracy as subdomains are refined (step-function approximation)

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 predicted functional values are near 4% and the run-time errors are under 1%. The system prediction errors follow the component approximation errors: in the lower half of the table, halving the subdomain size reduces the overall prediction error by about half. However, the measures that look at each subdomain ("Max" and "> 5") 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 III. The successive lines in Tables II and III represent the same subdomains in each component approximation. The final count of system subdomains in corresponding rows is larger in Table III because 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 III, which repeat the 6th and 4th rows from Tables II and III 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. In terms of the effort required of a component developer, the piecewise-linear approximation is thus about 20 times better. Table III also displays the anomaly of persistent errors in a few subdomains in the first three rows, to be discussed below.

System sub-domain count	Component measurements		System predictions and measurements							
	rms % error		Functional % error				Runtime % error			
	Function	Runtime	Overall	Max	Mean	> 5	Overall	Max	Mean	> 5
13	3.78	0.00	10.52	9.62	4.06	5	0.04	4.04	0.32	0
26	1.42	0.00	3.28	5.94	2.06	2	0.01	0.13	0.01	0
52	0.33	0.00	1.18	7.97	0.68	2	0.00	0.05	0.01	0
96	0.12	0.00	0.44	1.66	0.13	0	0.00	0.02	0.00	0
191	0.02	0.00	0.06	0.22	0.02	0	0.00	0.01	0.00	0
384	0.00	0.00	0.01	0.04	0.01	0	0.00	0.00	0.00	0
775	0.00	0.00	0.01	0.01	0.00	0	0.00	0.00	0.00	0
Comparison between step-function and piecewise-linear approximations										
128	0.52	0.22	2.01	4.41	0.88	0	0.43	7.78	0.20	1
96	0.12	0.00	0.44	1.66	0.13	0	0.00	0.02	0.00	0

Table III. Prediction accuracy as subdomains are refined (piecewise-linear approximation)

Tables II and III show the component-measurement errors steadily decreasing with smaller subdomains, which is a consequence of approximation theory for real-valued functions[9]. The theory is validated in the case study by the corresponding steady decrease in the system-prediction error. Furthermore, there is a nearly linear relationship between the approximation error in the component measurements 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 II have been omitted from the upper curve as

[9] Pathological cases could be constructed with everywhere discontinuous functions that would not behave so well, but there are only eight discontinuities in the case-study components.

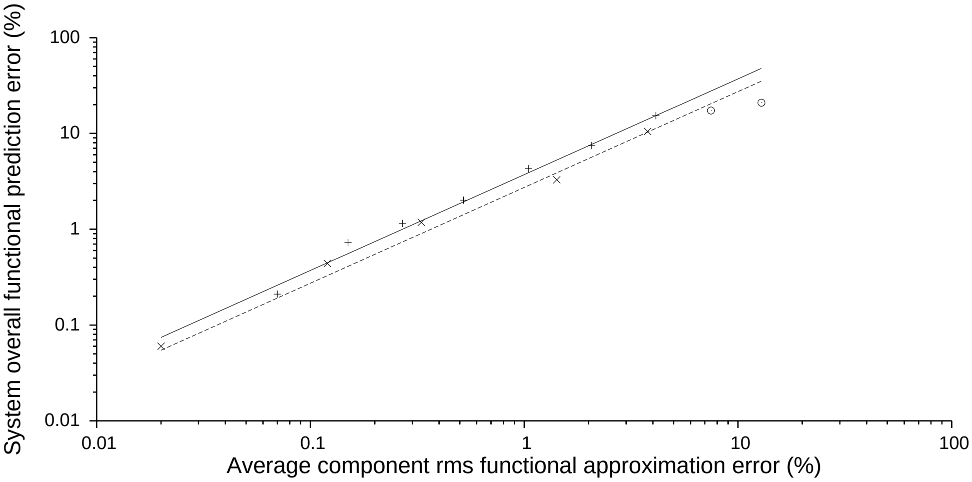


Fig. 10. Prediction error as a function of approximation error (upper curve: step-function; lower: piecewise-linear)

outliers (circled points). The proportionality constant grows with the complexity of the system being synthesized, being bounded by the number of synthesis steps, or roughly the number of components in the system.

The information in Tables II and III can also be 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.

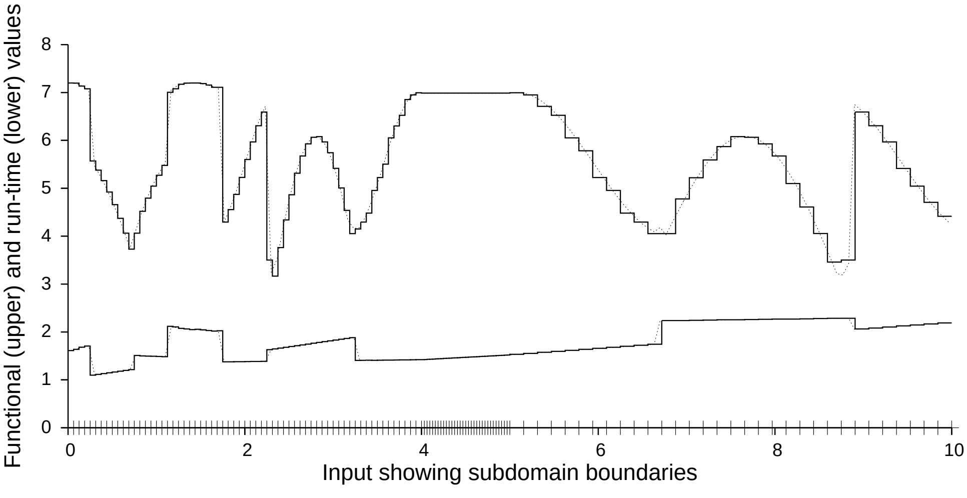


Fig. 11. Step-function-approximation predictions (solid line) and measurements (dotted line)

Figure 13 displays a small region of Fig. 11, showing a subdomain in which inaccuracy persists—the anomalous behavior in Tables II and III. The explanation is that predicted behavior can only change at subdomain boundaries. If there is a

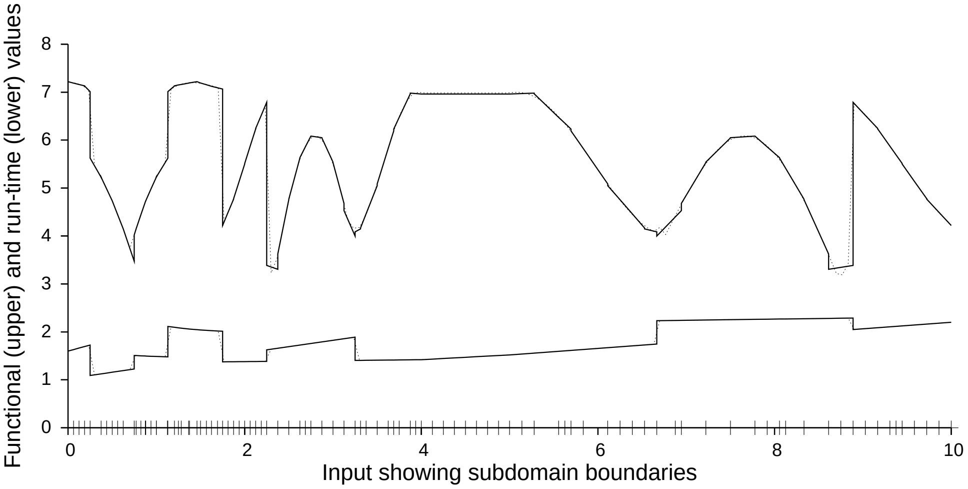


Fig. 12. Piecewise-linear-approximation predictions (solid line) and measurements (dotted line)

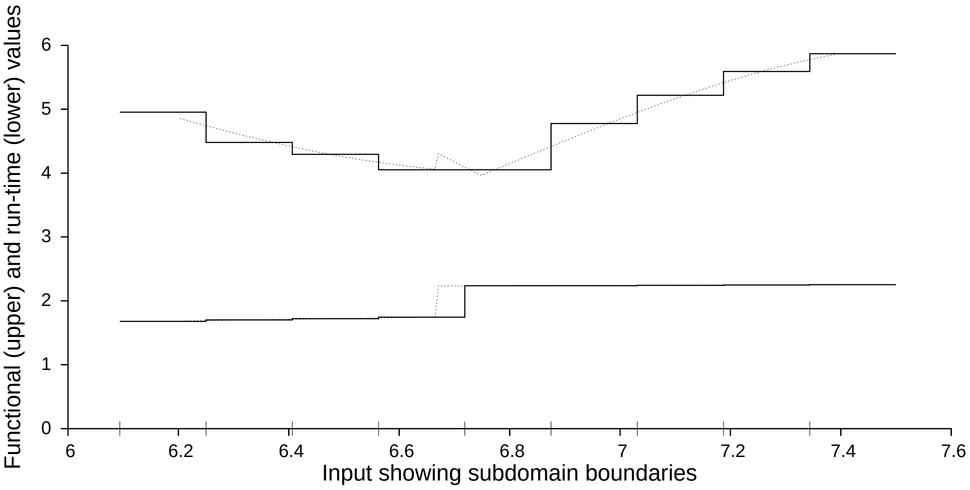


Fig. 13. Detail of Fig. 11 showing subdomain error

rapid change in actual behavior (a discontinuity in the figure) within a subdomain, the prediction cannot track it. Halving the subdomain size may only recreate the same (or worse) relative error in a smaller subdomain. However, such errors are confined to subdomains that occupy a smaller and smaller part of the whole domain, so the overall error decreases.

In Fig. 13 the discontinuity near 6.7 is an emergent system property that arises from a *combination* of components. Anomalous system subdomains are a residual form of the 'non-compositional' nature of testing discussed in Section 2. Component developers working in isolation can do nothing to mitigate the effect except improve the approximation of catalogue descriptions everywhere on components' domains. However, a component's discontinuities themselves usually create system discontinuities, and if the subdomain boundaries in the catalogue description are placed at the points of discontinuity a corresponding system anomaly does not occur. Developers should use particular care in creating subdomains for components that will be used as conditionals. A conditional's discontinuities are the points at which it switches between *true* and *false*, and a misplaced subdomain boundary leads to a subdomain with an 'average' value that is neither. The tools arbitrarily assign a binary value to such subdomains, but the prediction error that results can be out of proportion to the subdomain size, since it will cause the wrong component of two alternatives to be selected.

The insight that composition of subdomain-measured properties may have unpredictable emergent consequences requires all of the context of this experiment: a simple theory, supporting tools, and components chosen to force difficult cases to occur.

4.2 Subdomain-refinement Experiments with State

It was not difficult to select a typical example system for the stateless case study reported in Section 4.1—all stateless examples we tried exhibited similar features. The theory is taxed most when component output behaviors are discontinuous or rapidly changing, so we combined such components in a structure using all the possible system constructors. It is more difficult to devise a representative example with state because of the many ways local states might be used in a system. Making most of a system's components stateless produces the most comprehensible examples with graphical results (e.g., Fig. 7 in Section 3.2.2); but then little of the state theory is used.

Small case studies in which several components have state are unlikely to be representative because of the myriad of ways state is used. So instead of a single example, we present two that model common uses of state:

(1) Iteration to achieve convergence;
(2) Use of 'modes' and mode interaction.

These uses can be modeled only crudely within the restrictions imposed on components, but the exercise holds some surprises.

Random input sequences are the correct ones (as opposed to systematic sampling) to use in measuring component catalogue entries and in validating system predictions, because infeasible states are omitted. In graphs like Fig. 3, these states don't appear—rectangles are missing. Unfortunately, random input sequences may fail to exercise hard-to-reach but feasible states. It is an unsolvable problem to determine whether an omitted subdomain will never be sampled by any input sequence or whether some untried sequence would reach it. At the component level, as a part of verification, the tester should check that state subdomain sampling agrees with the domain of the component's specification[10]. At the system level, the same unsolvable problem arises: In an attempted validation, some system subdomains may not be sampled.

The reverse can also occur in validation: Some system executions can fall outside all of the calculated subdomains. In this case the fault always lies with some

[10] The process may be difficult, for one thing because the specification itself may have unsuspected infeasible states.

component catalogue entry: a subdomain there was not sampled, the component tester judged it infeasible, but was wrong, as values arising in system execution subsequently show. When this occurs in a system-synthesis calculation the quick fix is to redo the component sampling systematically. When there are infeasible states, this may waste a lot of time, and it distorts the measurement of component approximation errors (because errors in infeasible states should not count). As discussed in Section 6.1, at system level systematic sampling doesn't make sense, so the only remedy for apparently missing states is more extensive random input sequences.

4.2.1 *Iteration to Convergence.* A common use of state is in a loop whose body repeats until some kind of convergence is obtained. Our example has a test component C_t that uses its state to remember the previous value returned by a body component C_r. C_t terminates the loop when the next iteration returns a value close enough to the previous return. C_r uses its state to remember the term of a geometric series which it accumulates. The system

$$\texttt{while}\ C_t\ \texttt{do}\ C_r\ \texttt{od}$$

thus crudely models numerical computations like Newton-Raphson iteration.

Table IV shows what happens as the component subdomains are refined by repeatedly splitting them in half, starting with about five subdomains in each dimension for each component. In Table IV the component measurements are systematic

		R-m-s % errors											
Number of system		Component measurements						System predictions and measurements					
subdomains		output		run time		state		output		run time		state	
total	feasible	ave	max	ave	max	ave	max	ave	max	ave	max	ave	max
150	20	21.5	54	5.3	34	21.0	54	33.5	81	20.7	38	351	665
1200	59	3.2	54	0.0	0	12.6	54	31.7	162	8.8	55	179	566
9600	144	8.4	141	1.7	35	7.4	54	18.6	145	6.1	53	98.3	557
76800	338	4.2	141	0.8	35	4.2	54	11.0	138	4.5	53	55.1	505
422400	733	1.3	71	0.3	28	2.4	54	3.9	62	1.5	26	25.4	398
3328000	1203	0.7	75	0.2	28	1.3	54	2.7	64	1.2	28	13.4	411

Table IV. Measurement and prediction errors for the system `while` C_t `do` C_r `od`

with nine samples/subdomain; values for C_t and C_r are averaged in the 'Component measurements' columns. System measurements were made with 60 random input sequences containing 1546 points. The system state is a two-fold cross product of local states from C_t and C_r; the error in the last state column of Table IV is the magnitude of the vector sum of the errors in its two parts. The columns labeled 'max' are the error in the worst subdomain. The discrepancy between the weighted average error over all subdomains ('ave' columns) and the maximum in the worst subdomain indicates that many subdomains were badly approximated. For example, in row 3 of Table IV, the functional prediction whose weighted average error was 18.6% was worse than this in 39 out of 144 subdomains.

Table IV generally shows improvement in the component measurement accuracy, leading to improved prediction accuracy. However, the improvement is much slower than in the stateless case (Table II in Section 4.1), the state prediction never becomes very good, and there are persistent large measurement/prediction errors in some subdomains (further discussion below). The number of system subdomains grows by about eight times in each successive split (2^2 more subdomains $\times 2$ components), but almost all of these represent state combinations that never occur. (In the last two data rows of the table, some infeasible state subdomains were removed by hand to shorten the calculations.)

The component measurements in row 2 (1200 subdomains) are anomalous because there is a lucky coincidence between subdomain boundaries and the switch between *true* and *false* by C_t. The boundaries do not usually line up, as Fig. 14 shows for the more typical measured behavior of C_t in row 3 of the table. The 38 mid-level rectangles are the subdomains in which there is a 141% error. For such 'mixed' cases the tools choose the closest one/zero value, zero in Fig. 14. In row 2 of Table IV mixed decisions luckily did not arise, so the functional and run-time component measurements in row 2 are better than in other rows. The 'mixed' behavior does not disappear for smaller subdomains as explained at the end of Section 4.1.

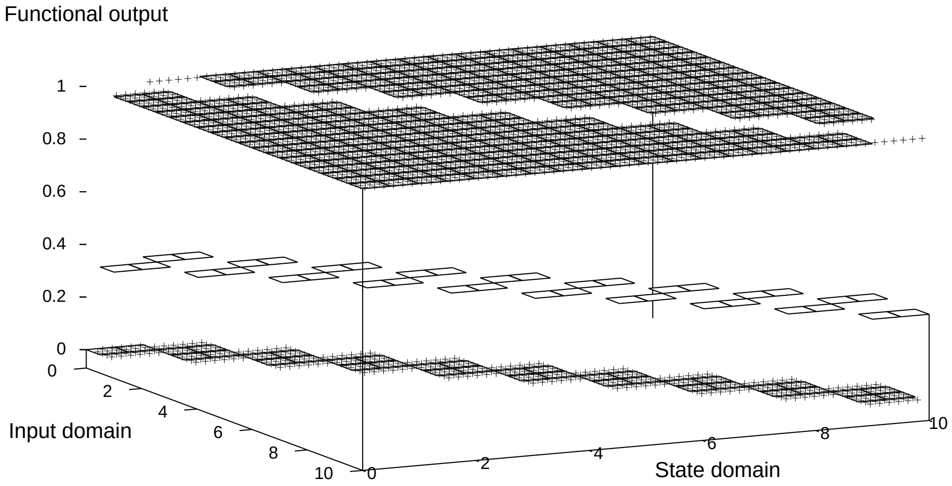


Fig. 14. Approximate (rectangles) and measured (crosses) output for loop-test component C_t

It remains to explain the persistent large errors in some subdomains and the overall poor prediction of system result state (last columns of Table IV). In these experiments, as in all our case studies, we checked the correctness of the CAD tools in each instance by comparing their prediction to the table-lookup 'execution' of a system formed from the catalogue entries of its components, as described in Section 3.3. The check failed for conditionals when either branch component has state, and in loops (as a consequence of unrolling into conditionals) when the body component has state, as in the system of Table IV. This anomaly arises from an inherent limitation of step-plateau approximation (and not from mistakes in the CAD implementation as in many other cases!). In calculating the state values of the equivalent component for a conditional, values must be supplied for local states in the 'other' branch. For example, a subdomain in the equivalent may come from the THEN branch, but part of its result-state value arises from state in the ELSE branch. The ELSE is not involved in the composite output or run time or the part of composite state that comes from the THEN component, but a value must be calculated for the state part that is local to the ELSE. The correct calculation is identity, but a true identity value is not available in the step-plateaus. The best that can be done is to fill in some value that lies in the input state subdomain; the CAD tools use the midpoint. This usually disagrees with the table-lookup execution of components that is literally leaving the state of the unexecuted branch alone, a true identity. The anomaly could be corrected as described in Section 6.1, but in Table IV it accounts for the large state prediction error. In subsequent series compositions, these wrong states can either disappear (when they happen to fall in the correct subdomain after), or can cause growing errors, not only in state, but in output and in run time, because they go to the wrong subdomain. Loop synthesis is the worst case, because the calculated state of the body can drift badly.

It is evident that the approximation theory is not useful for predicting the results of numerical convergence: using over 3 million subdomains to get 3% average functional accuracy instead of a half-dozen iterations of actual code is no bargain. The implication is that such iterations should be encapsulated in a single component, not implemented with an inter-component loop. In this case component independent development is a false goal.

4.2.2 *Modes and Mode Interaction.* A case study used to study the relationship between unit- and system testing [Hamlet 2006] can be adapted as a refinement experiment. It uses two components with state:

C_e, *an 'editor' component,* uses its state to model two basic ideas of a text editor like the Unix 'visual editor' (vi). First, an editor has two basic modes, one for input in which most inputs are just stored, and one for commands in which most inputs cause editing actions of varying complexity. Second, in command mode, state is also used to tailor commands, for example to remember the last string argument so that it need not be retyped.

C_c, *a control component,* is intended to act as a front-end to C_e, using its state to remember several modes:

—Shadow of C_e's input/control mode, so C_c knows what C_e will be doing. (It cannot of course access the local state of independent component C_e.)

—Mode to invoke C_e, but C_e is not permitted to change its input/command state, nor to change its tailoring parameter.

—Mode to force C_e to toggle between input/command state, split into submodes to allow only three such changes.

—Dialog mode in which C_e is not involved.

Stateless components C_d and C_a are also needed: C_d handles the dialog; C_a is glue code that filters inputs to C_e to prevent it from taking actions not desired by C_c.

[Hamlet 2006] gives complete details of the behavior of these components. Here we display only the output behavior of C_e provided by the tools, to show how the editor-like features are modeled. In Fig. 15, negative states (at the front) represent 'editor input' mode in which neither input nor state affects the behavior; positive states (at the back) are for 'editor command' mode, with complex input-dependent output and a state-dependent aspect to the output (the linear drop-off toward state 0 in the middle of Fig. 15), which models the tailoring of commands using a state parameter. States in [-0.5,0.5) are infeasible, modeling that not all parameter values are allowed.

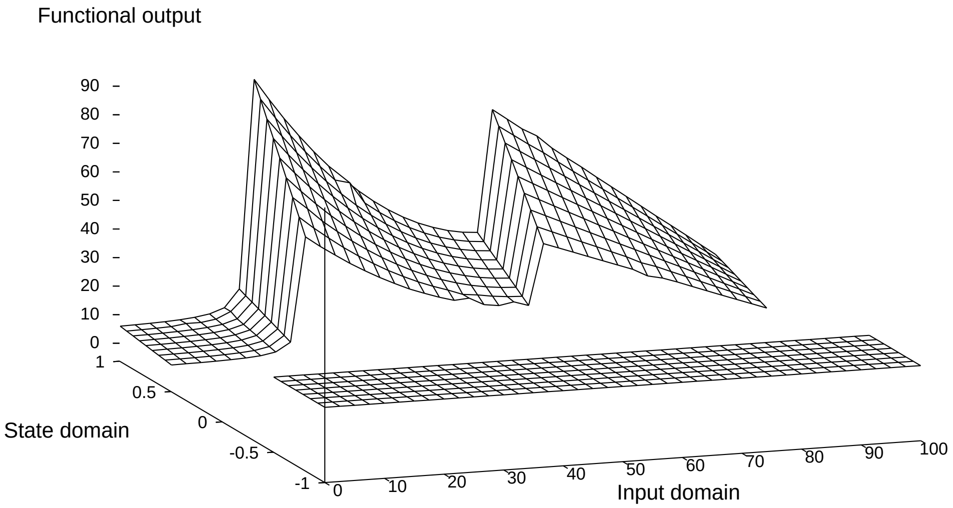


Fig. 15. Output behavior of C_e modeling text-editor-like behavior

The system structure of the case study is:

$$\texttt{if } C_c \texttt{ then } C_a;\ C_e \texttt{ else } C_d \texttt{ fi}$$

[Hamlet 2006] describes the expected system behavior with eight composite system modes that are cross products of local states in C_c and C_e. Here we are only interested in how well the behavior of a complicated system can be predicted.

Table V shows the effect of subdomain refinement, in the same format as Table IV. Random input sequences were used for the component measurements; even with up to 300 sequences and 45791 points a few feasible subdomains were missed for C_e in the last two rows of Table V, which led to some system subdomains without predicted values (4 of them in the 2nd-last row—0.3%; 57—0.2%— in the last row).

		R-m-s % errors											
Number of system		Component measurements						System predictions and measurements					
subdomains		output		run time		state		output		run time		state	
total	feasible	ave	max	ave	max	ave	max	ave	max	ave	max	ave	max
4368	208	4.5	46	0.9	43	3.1	12	6.1	49	1.6	24	6.0	10
34944	415	2.8	71	0.5	52	1.8	8	3.7	74	0.9	38	2.9	5
10192	415	2.0	13	0.1	16	2.1	9	3.8	97	0.9	43	3.0	5
81536	811	1.0	8	0.1	17	1.0	6	1.6	58	0.2	23	1.5	3
652288	1512	0.4	4	.02	16	0.5	3	0.7	32	.02	2.7	0.7	1
5218304	2470	0.2	3	.01	17	0.3	2	0.3	7	.01	.2	0.3	1

Table V. Measurement and prediction errors for an editor-control-like system

System validation measurements used 120 random input sequences totalling 7217 points.

Rows 2 and 3 of Table V are bracketed together to show the effect of hand tuning component subdomains. Row 1 was obtained with about 10 subdomains for each component in each dimension and no attempt was made to match them to component behavior. Row 2 has these arbitrary subdomains each split in two. Row 3 starts from row-1 subdomains, but instead of mechanical splitting, they were adjusted by hand to reduce component measurement errors, using tool-provided graphs like Fig. 3 and lists of errors in each subdomain. This effort was successful as the bracketed rows show. With the hand-tuned subdomains in row 3, the prediction results are about the same as in row 2, but with less than a third as many subdomains for calculation. Rows 4-6 of Table V were obtained by successively splitting all subdomains in half, starting with the hand-tuned ones of row 3.

In Table V the weighted average prediction error nicely tracks the reduction in components-measurement errors, but again errors remain large in some subdomains. This anomaly is less severe than in Table IV; for example, in row 4 of Table V, the functional prediction average error of 1.6% has a maximum of 58%, but an error over 5% occurs in only 36 out of 811 subdomains (4%). The explanation for such persistent errors is thus probably the same as that for the stateless system in Section 4.1. However, the difficulty of displaying four-dimensional graphs makes it harder to be sure about the cause.

4.3 Summary and Critique of Refinement Case Studies

Testing of software in the ideal CBSD paradigm where components are strictly separated from systems is a way to investigate the question of unit- vs. system testing. If CBSD tools like ours work, unit (that is, component) testing could assume a far greater role than it has today. In the extreme case, algorithms for synthesizing an approximate system and calculating its functional and non-functional properties allow system testing itself to be eliminated. In this extreme there would still be system requirements/specifications, but instead of executing assembled code against them, the calculated predictors of system behavior would be the 'implementation' for verification [Hamlet 2006]. For the tools described here, the verification would take the form of 'executing' by table-lookup the equivalent component that approximates the complete system behavior. The advantage over executing real system code is speed and strong tool support that is independent of programming languages and operating system platforms.

Another way to phrase the unit- vs. system testing question is: Is testing really non-compositional as is usually supposed? To what extent can everything about assembled system properties that could be learned by testing the system, be instead learned from only component measurements and calculations?

Some observations about the experiments presented in Section 4:

Simple case studies. The case-study systems are all small and they use artificial components. These components and combinations were chosen to tax the capabilities of the theory and tools, using experience gained from pilot experiments (Section 3.4).

Complex behaviors. Even starting with a healthy respect for how complicated software is, it is surprising how difficult it is to grasp intuitively the behavior of the simplest components and systems. Explicit graphs like Fig. 6 show unexpected behavior; when there are no graphs as in the studies of Section 4.2, it brings home how hard it is to know much about what software is really doing.

Quality of unit testing. The r-m-s deviation of a measured approximation from the actual behavior of a component (Section 3.1) is a precise quality measure of the unit test from which it is taken. It validates this metric to show, as Section 4 does, that it can be used to accurately predict system properties. In contrast, the usual metrics of test quality are weak surrogates. There appears to be no theoretical or sound experimental evidence validating unit-test coverage measures of test effectiveness, for example. If indeed a good unit test must be able to serve as the basis for system predictions, then the failure of unit-test methods in practice is explained, because they use a very few ill-considered subdomains.

Input, output, run time, and state. A complex interdependence exists between predictions of output, run time, and state. All predictions should be dominated by functional-output predictions, since in series composition the second component receives its input as the first's output. There is no such general cascade effect involving state prediction, since component states are local. However, errors in state prediction accumulate for the body component in a loop, making loop predictions the least accurate. As a consequence of these dependencies, it is possible to have perfect component approximations of one property compromised by the others. For example, the system run-time predictions in Table III are not perfect although the component run-time measurements are perfect.

Tuning subdomains with state. It is much harder to adjust subdomains to better capture the behavior of components with state than when they are stateless. Partly the reason is that useful graphical presentations are much harder to give (a consequence of the quadratic rise in descriptive data for the state case). But interactions in behavior between input and beginning state can also make it impossible to capture boundary behavior with rectangular subdomains, since the boundary may be an arbitrary curve through (input×state) space. Fig. 14 is a simple example where this curve is a diagonal line.

Tool validation. Artificial case studies are only weak validation of a theory. Reality can't be simplified without perhaps omitting its very aspect that would kill the theory. But the experiments of Section 4 and many others less organized provide high confidence in the tool implementations. At each refinement step of every experiment, a comparison was made between the calculated system and table-lookup 'execution' of the approximated components in the system configuration as described in Section 3.3. This has the effect of using the tools on a series of more and more complex components that are each perfectly approximated (because the approximations are being treated as if they *are* the components for execution). An observed zero-error system prediction verifies that the tools are working correctly. It will be no surprise that many mistakes in the tools were exposed and corrected in this way.

This work was begun as an exercise in subdomain-testing theory [Hamlet et al. 2001] without any idea of implementation. So the most surprising result is that it is possible in principle to algorithmically calculate what a system will do from component measurements. The extreme restrictions on component interfaces are necessary to the algorithms, but that is one accepted path to theoretical explanation. The initial theory did not include state (as most testing theory does not to this day), so it was an additional surprise that the addition of state-processing to the tools was straightforward.

For stateless components the theory and tools work very well. The only thing that seems to matter is rapid or discontinuous change in the functional behavior of components. When subdomains are sized and placed to capture that behavior, accurate predictions result. Good graphical visualization support is easy to provide and very helpful in understanding what happens.

When there is state, no simple summary is possible. The use of state to implement a handful of 'modes' is really just a way of packaging a handful of stateless behaviors in one. Once the state subdomains accurately isolate each mode, predictions are accurate if the input subdomains handle discontinuities as above. On the other hand, interactions between input- and state functions may never be captured adequately. The more components have state, the harder it is to understand the cross-product system state behavior and the more poorly the approximation and synthesis algorithms perform. In particular, state combinations where a conditional-test component has state and so does a component it controls may not be well captured by the synthesis approximations.

When the theory fails to give accurate predictions, there are two interpretations: The first, obvious one is that theory is useless and sensible people should ignore it (this theory, anyway). But a second interpretation is that this theory describes how well unit-test quality translates to system quality. So when a particular case fails, that case is one in which unit testing is not valuable. Extending this insight, those units themselves may not be of value—it would be better to combine the offending combination into a single component. The tradeoff is between how difficult it is to test single components well (when too many have been coalesced) and how meaningless good component tests are (when there is too much separation).

The most interesting observation to arise from the experiments of Section 4 is the existence of emergent system test properties such as prediction errors that persist in some subdomains (Section 4.1, Fig. 13). These seem to be the non-composable nature of testing re-appearing at the subdomain level when subdomains banish it at the domain level. In the ideal CBSD process there is nothing a component designer can do about failures of the theory that only emerge for particular system designs, except perhaps test to a far higher standard than seems needed or wise. In the experiments of Section 4 we strictly adhered to the ideal: we used measurements of system properties only for validation, never to adjust component measurements. For example, an emergent system discontinuity could have been traced back to the component combination that created it, and the subdomains of those components adjusted to create better system subdomains near the discontinuity. But in practice, a system designer might make the best of both worlds by breaking slightly with the ideal. A few rough actual system tests will indicate how good the CAD predictions are. If they are accurate, design can proceed using them. If they are poor, the problem can be traced to an offending component combination (as in Section 5.2 to follow) and those components can be replaced with better ones. Or, components can be sent back to their developers for further directed testing peculiar to emergent properties of this particular system.

5. USING COMPONENT CATALOGUE ENTRIES AND CAD PREDICTIONS

When a system designer decides to use a certain collection of components in a particular design, working with ideal CBSD tools is the same as working with code except that: (1) The tools can be much faster and easier to use than code executions, and (2) Everything is only an approximation to actual behavior. Insofar as the designer would like to learn anything using testing methods, it can be done approximately without resorting to code. The ideal CBSD paradigm protects proprietary information about components, since not even their binary forms are made public.

5.1 Functional vs. Non-functional Predictions

System testing requires an oracle, usually a human being reading a system specification, to judge the success or failure of test executions. In some cases, the approximate CAD-synthesized tables can be usefully compared to an oracle's judgement. It is possible to get a good sense of the general shape of a piecewise continuous function by looking at graphs such as Figs. 6 and 7. However, except for scientific applications in which numerical results have specified error bounds, the CAD-calculated functional values are likely to always be judged failures, even though they are (say) within 1% of what actual code correctly does. However, even rough functional approximation suffices for the prediction of non-functional properties like run time. Non-functional specifications are often less precise than functional ones. For example, run time may be required to be within a given response time, or reliability to exceed a given value. If there is some leeway in such specifications, it may be enough to absorb the calculation errors in the CAD tools, and permit a valid prediction that a non-functional requirement has or has not been met by a design using particular components.

For example, in Fig. 11, suppose that the response time is required to be less than 3. Then knowing the catalogue-entry error of .22% (Table II, row 6), and that in a six-component system the CAD calculation might stretch that error by about 5 times (Fig. 10), the predicted run-time values might be wrong by about 1%. From the graph, the largest predicted run time is about 2, so a prediction that the designed system will meet its response-time requirement seems safe, since $2 \pm .02 < 3$. Even in the much poorer results for systems with state, the run-time predictions are not bad: at around 10000 subdomains where the calculations are almost instantaneous, the run-time error is about 6% in Table IV and under 1% in Table V.

5.2 Interface Between Components

The tools that execute a 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 system developer to study what happens at the internal system interfaces.

A more detailed variation of the 'composability' problem for testing theory mentioned in Section 2 is profile distortion between components. The operational profile of input data presented to a system is seen by the first component, but as information passes among the components the profile is distorted. Adequate testing of a component buried in the system structure should take account of this profile it sees in place, but of course the *in situ* profile cannot be known until the system design is complete and all components are in place. Unfortunately, profile distortion in a system is substantial even in the simplest cases. For example, if the system of Fig. 5 described in Section 3.2.1 is given inputs from a uniform profile (density 0.04 over each of 25 subdomains), the profile seen by C6 (the component in the *false* branch of the conditional in Fig. 5) is shown in Fig. 16. There is no way that the component developer can know that this would be the appropriate test profile;

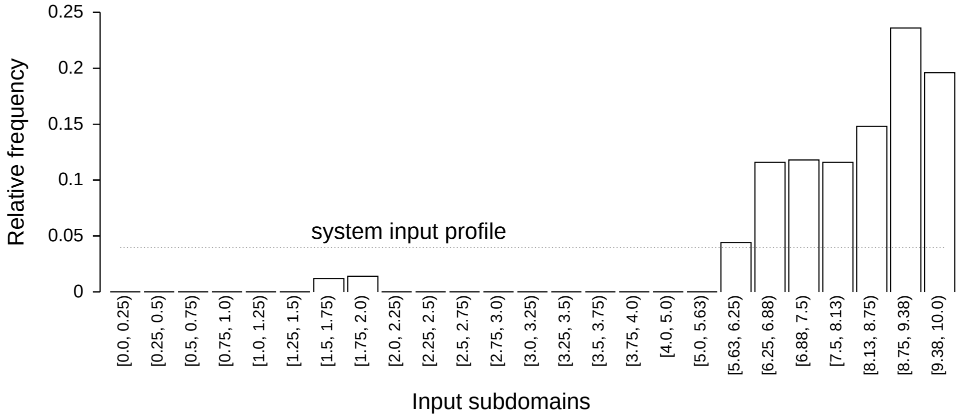


Fig. 16. Profile seen by component C6 on uniformly distributed input to system of Fig. 5

using a uniform test profile for C6 would be badly wrong. For example, the density seen by C6 in [8.75, 9.38) is about 0.25, not the uniform 0.04. Profile distortion is an elementary consequence of using components in series, but concrete examples such as this one bring it home.

It is subdomain testing that solves the problem of how to test a component without knowing its eventual use. No matter how the *in situ* profile weights inputs, they have been sampled. The payment for this generality is effort wasted in particular cases: For example, in Fig. 16 all the component tests in subdomains [0, 0.25) – [1.5, 1.75) and [2.0, 2.25) – [5.0, 5.63) are wasted for the trial system. The finer the subdomain division, the better an arbitrary profile can be handled, but the more effort may be wasted.

An extreme version of profile distortion occurs when one component's output goes outside the catalogue-entry input domain for a following component. When this happens in executing a real system, there is usually a catastrophic system failure. The Ariane-5 flight-control software failed [Lions 1996], for example, when a component sent a symbolic error code as output to a routine expecting a short integer. The CAD tools, by calculating for all system-input subdomains, must necessarily detect any such problem, and the trace data allows the system designer to study the failure before system construction. There is a difficulty, however, in presenting the failure information to the CAD tool user. In a sequence composition of two components CA and CB, the CAD tool issues messages like:

```
Output 103.6 from CA subdomain [3,7) does not fall in any
CB subdomain.
```

The system designer can then examine the situation to discover whether CA is producing the wrong value on [3, 7), whether CB should have been able to handle 103.6, or whether the design itself is wrong, e.g., CA should not be in series with CB.

But suppose that such a difficulty arose[11] in the example system of Fig. 5, with the message

`Output 9.5 from` E_2 `subdomain [1.25,1.5) does not fall in any`
E_3 `subdomain.`

The message arises in the last step of the synthesis of Fig. 5 when the subsystem consisting of components C1, C2, and C3 (equivalent component E_2) is being composed in series with the subsystem of components C4, C5, and C6 (equivalent component E_3). The diagram of Fig. 17 presents the situation.

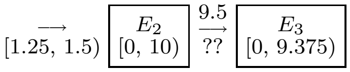


Fig. 17. Mismatch of two equivalent components in the synthesis of Fig. 5

The trace facility allows the tools to trace back to the original components that form the system by expanding the equivalent-component boxes. At the next level, expanding E_2 and E_3, we have Fig. 18. In Fig. 18 the designer can see that the

[11] To create the example, the subdomain [9.375, 10.0) was removed from the configuration file for component C6 in Fig. 5.

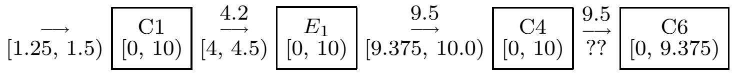


Fig. 18. Lifting of Fig. 17 by one equivalent-component level

mismatch is inside E_3 at the input to component C6.

The domain analysis of the example can be performed entirely by CAD tools, using no actual system executions[12]. It demonstrates the detection of a system design problem (mismatched component interfaces) "on paper."

5.3 A Reliability Application

When components are placed in series, their run times add (with the proviso that in the second component the run-time function sees an input that comes from the first component). For the non-functional property of reliability, the component failure rates should be measured, and the failure-rate complements (reliabilities) multiply where run times add. Taking the usual assumption that failure rate is constant in each subdomain [Musa et al. 1990], a stateless component's reliability description is a step function. The theory and tools described above can be used to measure component reliability and to predict system reliability.

However, the underlying reliability theory has an important deficiency. The low failure rates of high-quality software cannot be directly observed: in a practical amount of test time, reasonably good software does not fail. The best that can measured is an upper bound on failure rate and an upper confidence bound for the measurement [Hamlet 1994]. The component-synthesis tools therefore calculate only bounds on system reliability from bounds obtained by executing the components that make up the system, none of which is actually observed to fail. The theoretical predictions therefore really cannot be validated—all we have is prediction of a bound and experimental data that the bound is not violated, a much weaker result than the quantitative run-time validation experiments of Section 4. It was this difficulty that led to the use of run time for most of our studies. However, decomposition into subdomains exhibits some interesting reliability features.

Consider again the case-study system of six components used in Section 4.1. Table II was obtained with only a few samples per subdomain—enough to get fairly accurate functional- and run-time predictions. But with a low sampling frequency the reliability bounds are poor. Increasing the sampling to 300 random samples/subdomain in all components except 500 samples/subdomain in C5 and 700 in C6 improves the 90% upper-confidence-bound reliability to better than 0.9 (failure rates below 0.1) in all components. For the system of Fig. 5, the lower curve in Fig. 19 shows the predicted reliability for the system using 32 subdomains (the 4th line in Table II). The calculations are the same as those for the run-time curve in Fig. 11, but multiplying the step-function component failure rates instead of adding component run times. The lower curve in Fig. 19 predicts the

[12] As the trace facility is currently implemented, it does monitor executions, because the student who implemented it was not able to conceive of a 'static trace' as part of the CAD calculation. However, as explained in Section 3.3 these executions can be table lookups in catalogue entries, not actual component code.

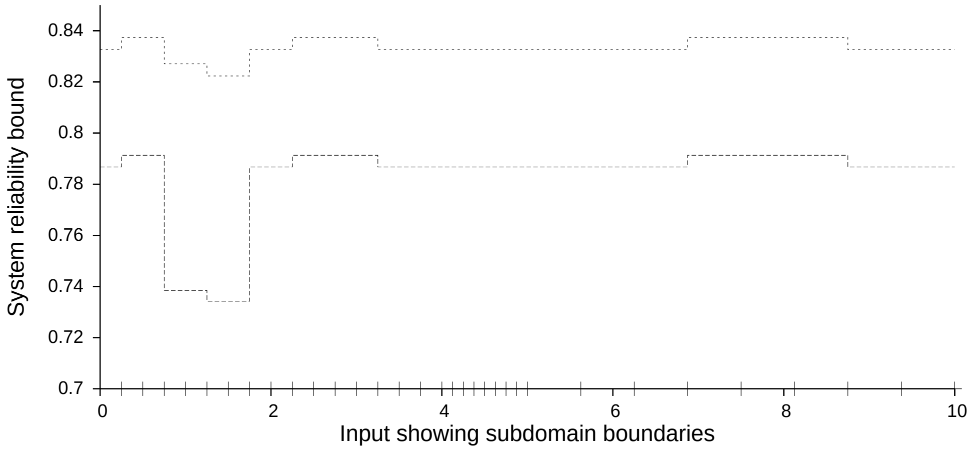


Fig. 19. Reliability predictions for the case-study system (upper curve: additional testing of C2)

poorest system reliability in the interval $[.75, 1.75)$. Guessing that this involves the loop construction, the test sampling of component C2 was increased to 8000 samples/subdomain[13], raising its reliability bound from about 0.94 to 0.99. The upper curve in Fig. 19 shows the system improvement.

The scenario described in this section, predicting the reliability of a system design using catalogue-entry values for its components, then making component adjustments to improve the result, is an example of design by trial. The system designer might request that the developers of component C2 test it more thoroughly, or might substitute another component whose catalogue reliability is better.

6. EXTENSIONS AND FUTURE WORK

Research-prototype tools are easy to extend and always in need of improvement. The present collection is coded in very pedestrian Perl; much of the work was done by high school students working under a summer mentoring program. Perl was chosen to make the learning curve shallow for short-term projects, which has been very successful but created tools that could use documentation and clean up. The current versions of all tools are available on-line [Hamlet 2007a], along with tutorial examples and many of the components used in this paper.

6.1 Issues for Tools that Handle State

The present tools that create catalogue entries for components with state can sample in two ways: Systematically in the two-dimensional (input×state) subdomains, or using random sequences of inputs. The former may record infeasible states as possible, but the latter can incorrectly miss a feasible state because not enough sequences were tried. A missed state can show up in system-synthesis calculations as an interface mismatch as described in Section 5.2. If there are only a few such

[13] Using the trace facility described in Section 5.2, the low reliability can be tracked to eight subdomains of C2, but an overall improvement results from retesting its whole domain.

omissions, it would be reasonable to simply ignore them in synthesis, rather than force the component designer to use systematic sampling. A compromise would be to mix the two by adding a few explicit subdomains to those that arise in random sequences. At the system level the tools try only random sequences of inputs because it seems unlikely that many of a huge collection of cross-product states will be feasible. (For example, in the last line of Table IV, only about 0.05% of the subdomains are feasible.) Further exploration of systematic sampling at the system level is called for.

In the initial CAD implementation including state, it was decided that only the step-function approximation would be implemented. This decision came from bad experience with the numerical stability of subdomain-splitting in the stateless piecewise-linear approximation CAD algorithms, despite their significantly better accuracy. It was not anticipated that this would in principle compromise the correctness of an equivalent component computed for a conditional (and hence also for a loop), as described at the end of Section 4.2.1. To fix that error, linear approximations could be implemented over both input- and state subdomains. The former follows what has already been done for the stateless case, but the latter is problematic: state values are often discrete and fitting linear functions seems inappropriate.

There is another subtle difficulty that would be resolved by implementing linear approximation in the input dimension for components with state. In a series composition whose first component is approximated by an input step function, the resulting equivalent component can have no subdomains but those of that first component. When the second component has complex behavior that can be captured only by careful choice of subdomains, yet the first component needs only a few, then although it looks to the components' developers like they have both done an adequate job, the synthesis calculation will be inaccurate: the second component subdomain detail will be washed out by the first component's simplicity. This happened with C_a; C_e in the system of Section 4.2.2. The effect doesn't show dramatically in our validation experiments, because all subdomains are being refined together. It does explain why in Table V the maximum predicted output errors remain high despite excellent approximations in all components.

Discrete state values are often thought of as 'modes' of a component or system, selecting a tailored behavior based on history rather than requiring input parameters to set the 'mode' on each execution. For this view it would be more appropriate to take the simplest state values to be integers rather than floating-point values, and a more natural model might allow multiple state values rather than require the state to be coded into a single value.

6.2 Application to 'Real' Software

The experimental exploration in this paper utilizes artificial components and systems chosen to stress the subdomain theory and provide fundamental insight into algorithms for system-synthesis prediction. The tools are general so long as the restrictions to single floating-point input/output/state values are observed. Although mathematical software sometimes does observe these restrictions, we do not believe that using it for 'real' examples will be informative. Real mathematical programs use sophisticated algorithms and extensive computations to get accurate results efficiently, but when viewed by the results alone, our components in Section 4 put more stress on the synthesis algorithms and reveal more about where they succeed or fail.

To experiment with more interesting components means lifting the restrictions to single numerical values. Using coding techniques from recursive function theory can turn a tuple of inputs into a single value, but allowing tuples directly only complicates the bookkeeping in tools, so that would be a first step. More important is to allow non-numeric types, notably 'string.' This raises hard questions about how to sample and approximate wider types, questions that testing theory has not addressed. For strings, one idea is to categorize strings by relevant properties (which might be derived from a specification) and to collect string inputs into subdomains using tuples of properties. For example, if string length, presence of upper case, and number of distinct member characters were three properties, consider the subdomain $S_1 = [5,9) \times \{1\} \times [1,6)$, that is, strings of length 5-8 containing some upper case and fewer than 6 distinct characters:

Strings in S_1	Strings not in S_1
`xxxXx`	`xxxxx`
`ABCDEE`	`ABCDEFGH`
`12244Z55`	`A1233333333`

Whether such subdomains are useful of course depends crucially on the property selection.

6.3 Concurrency

Many, perhaps most, component-based designs make essential use of concurrent execution. Often the parallel execution involves far-flung processors communicating over a network. The practical applications of these systems are of utmost importance, and a number of important issues such as multi-version programming (MVP) can only be discussed in a concurrent setting. The primary deficiency of the theory presented in Section 2 is that it makes no attempt to capture concurrency. Part of the reason for this omission is our goal of a simple theory—testing of concurrent systems is evidently not the place to start. But more important, it is hard to imagine a concurrent theory that is 'functional' in character, as the underlying Goodenough and Gerhart basis is. It may well be that extending testing theory is not a useful way to study concurrency. For testing itself, excellent tools have been devised to execute and monitor concurrent components, notable those of Kramer and Magee [Kramer and Magee 2006].

7. RELATED WORK

To the best of my knowledge, the work presented in this paper is new: Foundational issues in software testing theory are investigated using tool-driven experiments, in the context of software components and their composition. The testing theory that originated with Goodenough and Gerhart [Goodenough and Gerhart 1975] and the subdomain approach of Howden [Howden 1976] have previously not been applied to this context, nor have tools been implemented to experiment with testing theory in any context.

Other promising theoretical treatments of components use the pre- and post-condition formalism now called “design by contract” [Meyer 2000], model checking [Xie and Browne 2006], or bounded exhaustive testing [Dennis et al. 2006]. But these treatments do not apply to testing and they are less amenable to experiments. Perhaps the work closest to ours uses the Daikon tool [Ernst et al. 2001] that approximates programs using test data to induce assertions. However, Daikon uses testing as a device to investigate logic-based specifications rather than to look at testing itself; Daikon has not been applied to components and their composition. Meinke [Meinke 2004] has used approximation very like ours (and with the same restriction to numerical data) for seeking failures in stand-alone stateless programs.

8. SUMMARY AND CONCLUSIONS

We implemented tools to experiment with a fundamental, testing-based theory of software component composition. The theory approximates the behavior of components using subdomain testing, then uses these approximations to calculate approximate properties of systems built from the components; the tools measure the approximations, then make and validate system predictions.

We experimented with the tools using artificial components whose behavior could be easily adjusted to tax the capabilities of the synthesis algorithms. Theoretical predictions could often be made accurate by refining the test subdomains for their components. Good system predictions using stateless components turns on the discontinuities and rapid variation in those components’ functional behavior. These cause persistent prediction errors in some system subdomains, but as subdomains are refined the error subdomains contribute less and less. When components have local state, predictions are less good. An explosion in cross-product subdomains makes it difficult to understand why behavior is not being captured and it is too expensive to carry subdomain refinement very far.

Unit (component) testing is surprisingly good for obtaining system predictions without system tests. But to be trustworthy the unit tests must be painstakingly done. Today’s practice falls far short of the necessary standard.

Acknowledgements

Zheng Tu, Milan Andric, Ben Buford, John Christmann, Alex Corrado, Paul Draghicescu, Michael Plump, and Devon Gleeson worked on the prototype tools and early versions of the experiments.

REFERENCES

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

Dennis, G., Chang, F.-H., and Jackson, D. 2006. Modular verification of code with SAT. In *Proceedings ISSTA 2006*. Portland, ME, 109–119.

Ernst, M., Cockrell, J., Griswold, W. G., and Notkin, D. 2001. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. on Soft. Eng. 27*, 99–123.

Floyd, R. W. 1967. Assigning meanings to programs. In *Proceedings Symposium Applied Mathematics*. Vol. 19. Amer. Math. Soc, 19–32.

Goodenough, J. B. and Gerhart, S. L. 1975. Toward a theory of test data selection. *IEEE Trans. on Soft. Eng. 1*, 156–173.

Hamlet, D. 1994. Random testing. In *Encyclopedia of Software Engineering*, J. Marciniak, Ed. Wiley, New York, 970–978.

Hamlet, D. 2006. Subdomain testing of units and systems with state. In *Proceedings ISSTA 2006*. Portland, ME, 85–96.

Hamlet, D. 2007a. `www.cs.pdx.edu/∼hamlet/components.html`.

Hamlet, D. 2007b. Software component composition: subdomain-based testing-theory foundation. *J. Software Testing, Verification and Reliability 17*, 243–269.

Hamlet, D., Andric, M., and Tu, Z. 2003. Experiments with composing component properties. In *Proc. 6th ICSE Workshop on Component-based Software Engineering*, K. Wallnau, Ed. Portland, OR. `http://www.sei.cmu.edu/pacc`.

Hamlet, D., Mason, D., and Woit, D. 2001. Theory of software reliability based on components. In *Proceedings ICSE '01*. Toronto, Canada, 361–370.

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

Kramer, J. and Magee, J. 2006. *Concurrency: State Models & Java Programs, 2nd ed.* Wiley, New York.

Lions, J. L. 1996. *ARIANE 5 Flight 501 Failure Report by the Inquiry Board*. European Space Agency (ESA), Paris.

Meinke, K. 2004. Automated black-box testing of functional correctness using function approximation. In *Proceedings ISSTA '04*. Boston, 143–153.

Meyer, B. 2000. *Object-oriented Software Construction*. Prentice Hall.

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

Musa, J., Iannino, A., and Okumoto, K. 1990. *Software Reliability*. McGraw-Hill, New York.

Myers, G. J. 1979. *The Art of Software Testing*. Wiley-Interscience, New York, NY.

Xie, F. and Browne, J. 2006. Verification of component-based software application families. I. Gorton, G. T. Heineman, I. Crnkovic, H. W. Schmidt, J. A. Stafford, C. Szyperski, and K. Wallnau, Eds. LNCS 4063. Springer, 50–66.

Received Month Year; revised Month Year; accepted Month Year