DEET for Component-Based Software

Murali Sitaraman, Durga P. Gandi
Clemson University

Wolfgang Küchlin, Carsten Sinz
Universität Tübingen

Bruce W. Weide
The Ohio State University

Correspondence: murali@cs.clemson.edu
http://www.cs.clemson.edu/~resolve

This research is funded in part the U. S. National Science Foundation grant CCR-0113181.

What is DEET?

- DEET is Best Bug Repellent – *New England Journal of Medicine*, 2002.
- DEET is Detecting Errors Efficiently without Testing.

Correctness Problem and Well-Known Approaches

- Problem: Does the program do what is specified to do?
- Formal verification objective: Prove that it does, using static analysis.
- Testing (and runtime checking) objective: Find errors, i.e., find mismatches between specified intent and program behavior, through execution.

DEET vs. Verification vs. Testing

- DEET is a static analysis approach, like formal verification.
- DEET is intended for error detection, like testing.
- DEET has potential to serve as a cost-effective and efficient prelude to both testing and verification.

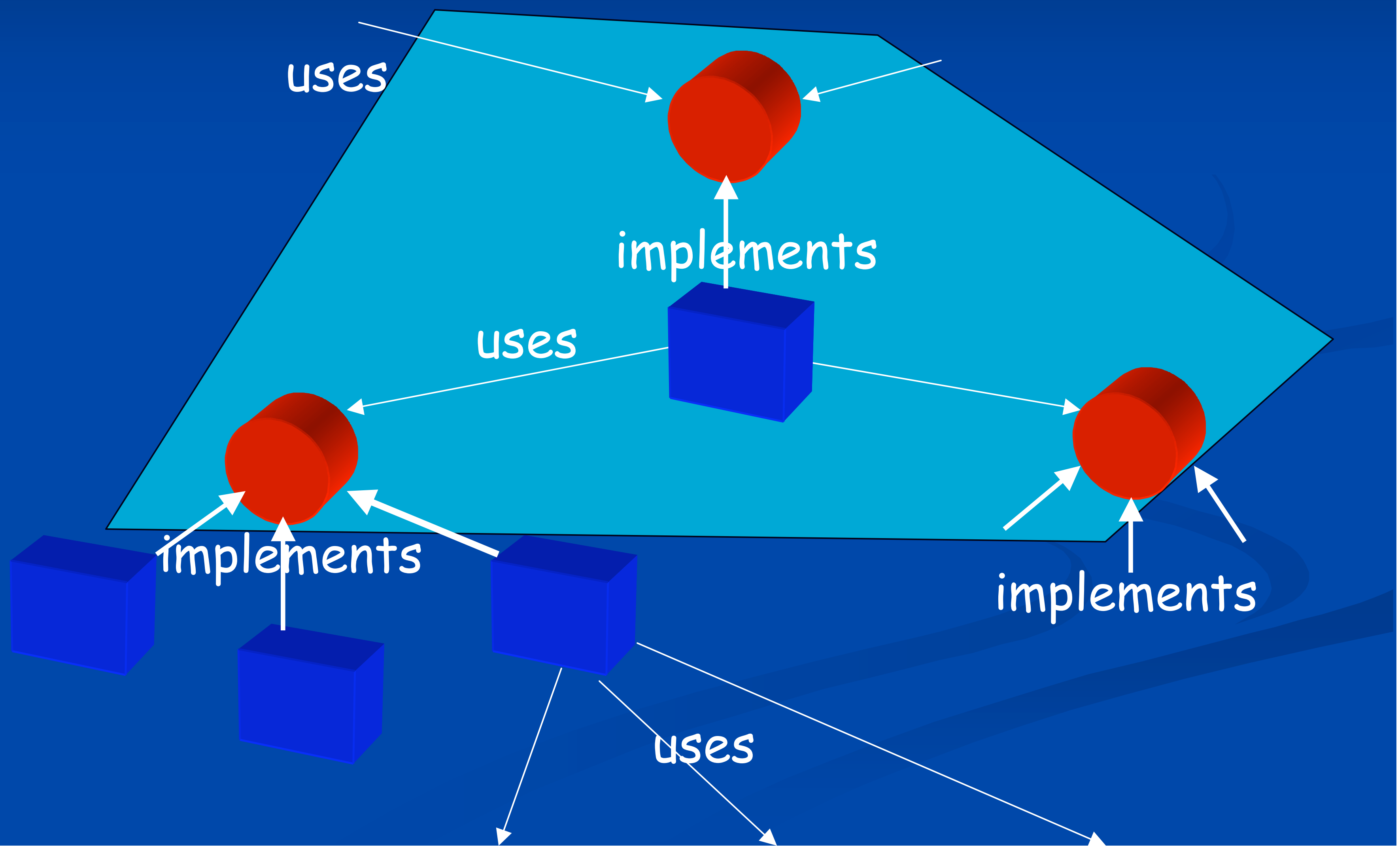
Benefits of the DEET Approach

- It can analyze one component at a time in a modular fashion.
- It does not depend on code or even stub availability for reused components; it can detect substitutability bugs.
- It is automatic and does not require manual input selection.
- It can pinpoint the origin of the error in a component-based system.

Contextual Differences Between DEET and Other Approaches

- Context of Alloy and ESC
  - industrial languages, such as Java
  - objectives are incremental based on current practice
  - minimal expectations of programmers
- Context of DEET
  - research language, i.e., Resolve
  - objectives are set in the context of software practice as it *could* be
  - a competent programmer hypothesis

Component-Based Software Using Design-By-Contract Paradigm



Ramifications of Contextual Differences

- DEET is a step towards meeting the larger objective of specification-based modular verification.
- In Resolve, components have specifications, and implementations are expected to have loop invariants, representation invariants, abstraction relations.
- Clean and rich semantics of Resolve allows variables to be viewed as having values from arbitrary mathematical spaces; references are not an issue.

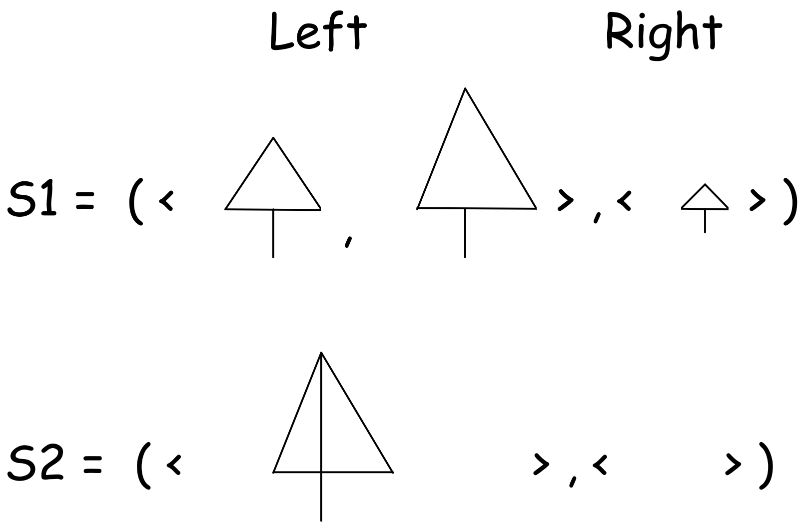
An Example

Abstraction in Specification

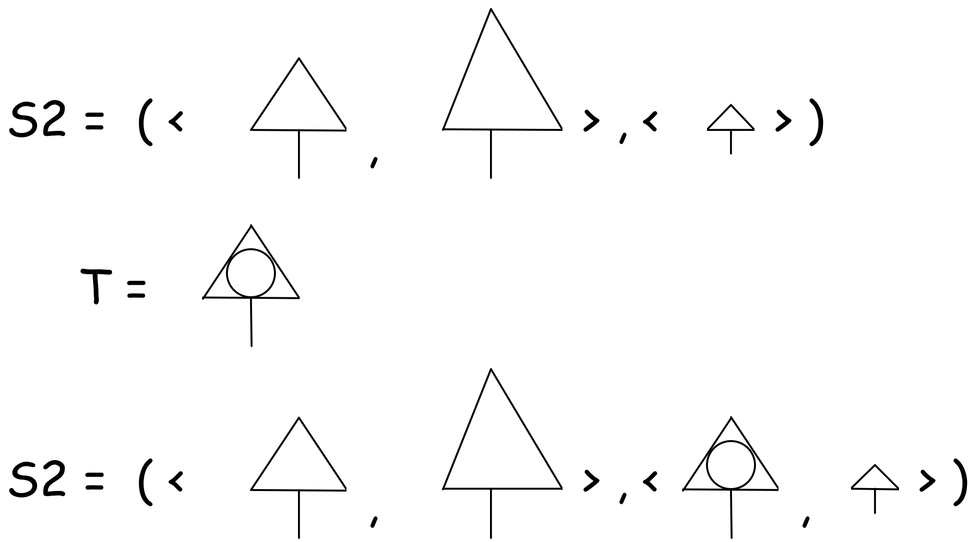
- Think of a List as a pair of mathematical strings:
  - A string of entries that are to the *left* of the "current position", and
  - A string of entries to the *right*.

- Initially, both strings are empty.

View of a List of Trees with Abstraction



View After Insert (T, S2)



Mathematical Modeling

```
Concept List_Template (type Entry);
        uses String_Theory, …;

    Type List is modeled by (
        Left: String(Entry);
        Right: String(Entry)
    );
        exemplar S;
        initialization ensures
          S.Left = empty_string and
          S.Right = empty_string;
    ...
end List_Template;
```

List Operations

```
Concept List_Template (type Entry);
        uses ...
    Type List is modeled by ...
    Oper Insert(E: Entry; S: List);
    Oper Remove(E: Entry; S: List);
    Oper Advance(S: List);
    Oper Reset(S: List);
    Oper Advance_To_End(S: List);
    Oper Left_Length(S: List): Integer;
    Oper Right_Length(S: List): Integer;
    Oper Swap_Rights(S1, S2: List);
end List_Template;
```

Design and Specification of Operations

Operation Insert(**clears** E: Entry; **updates** S: List);
  Ensures S.Left = #S.Left **and**
    $S.Right = \langle \#E \rangle \circ \#S.Right$;

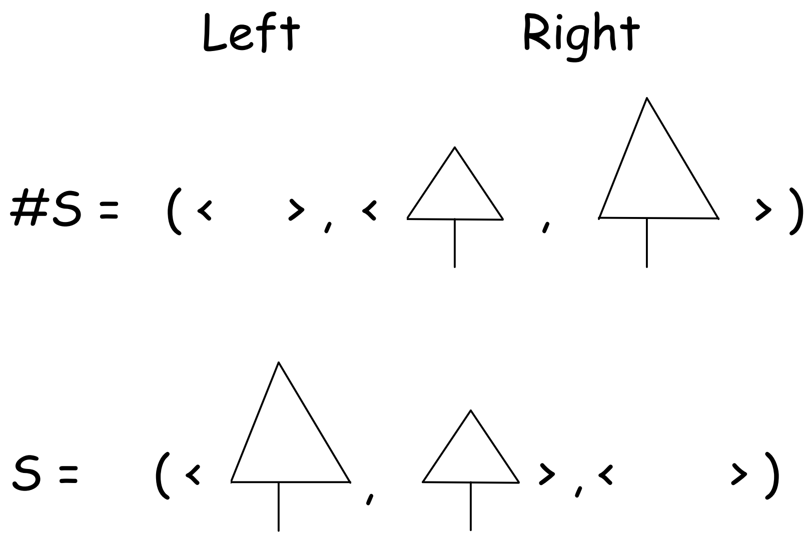
Operation Remove(**replaces** E: Entry; **updates** S: List);
  Requires $|S.Right| > 0$;
  Ensures S.Left = #S.Left **and**
    $\#S.Right = \langle E \rangle \circ S.Right$;

Part II: Erroneous Code Example

A Specification of List Reverse

Operation Reverse(**updates** S: List);
  Requires |S.Left| = 0;
  Ensures S.Left = #S.Right^{Rev} **and**
    S.Right = **empty_string**;

Example Behavior of Reverse



An Erroneous Implementation

```
Procedure Reverse (updates S: List);
        decreasing |S.Right|;
   Var E: Entry;

   If Right_Length(S) > 0 then
        Remove(E, S);
        Reverse(S);
        Insert(E, S);
   end;
end Reverse;
```

DEET Steps for Error Detection

Step 1: Verification Condition Generation

- What do we need to prove that the code is correct?
- What can we assume?
- What do we need to confirm?

Step 1: Verification Condition Generation

Procedure Reverse (**updates** S: List);
    decreasing |S.Right|;
  Var E: Entry;
0   Assume: $|S_0.Left| = 0$;
  If Right_Length(S) > 0 **then**
    Remove(E, S);
    Reverse(S);
    Insert(E, S);
  end;
5   Confirm: $S_5.Left = S_0.Right^{Rev}$ **and**
    $S_5.Right =$ **empty_string**
end Reverse;

Step 1: Verification Condition Generation

State	Path Condition	Assume	Confirm
0		$\lvert S_0.Left \rvert = 0$	
If Right_Length(S) > 0 **then**			
1	$\lvert S_0.Right \rvert > 0$	$S_1 = S_0$	$\lvert S_1.Right \rvert > 0$
Remove(E, S);			
2	$\lvert S_0.Right \rvert > 0$	$S_2.Left = S_1.Left$ **and** $S_1.Right = \langle E_2 \rangle \circ S_2.Right$	$\lvert S_0.Left \rvert = 0$ **and** $\lvert S_2.Right \rvert < \lvert S_0.Right \rvert$
Reverse(S);			
3		…	…

Step 2: Error Hypothesis Generation

- Conjoin assumptions and negation of what needs to be confirmed.
- Search for a counterexample.

Step 3: Efficient Searching for Counterexamples by Restricting "Scope"

- Restrict the "scopes" of participating variables, i.e., limit the mathematical values they can have.
- For variables of type Entry, suppose the scope is restricted to be of size 1.
  - Entry scope becomes: {Z0}
- For variables of type Str(Entry), suppose that the length is restricted to be at most 1.
  - The scope of String of Entries becomes:

    {Str_Empty, Str_Z0}

Step 3: Use Scope Restriction to Generate a Boolean Formula: Example

Boolean formula that corresponds to P1 = P0:

((S1_Left_equals_Str_Empty ∧
S0_Left_equals_Str_Empty) ∨
(S1_Left_equals_Str_Z0 ∧
S0_Left_equals_Str_Z0)) ∧

((S1_Right_equals_Str_Empty ∧
S0_Right_equals_Str_Empty) ∨
(S1_Right_equals_Str_Z0 ∧
S0_Right_equals_Str_Z0))

Step 4: Employ a SAT Solver to Search for a Solution

Set these to true

S0_Left_equals_Str_Empty

S0_Right_equals_Str_Z0

…

S5_Left_equals_Str_Empty

S5_Right_equals_Str_Z0

Set these to false

S0_Left_equals_Str_Z0

S0_Right_equals_Str_Empty

…

Efficiency of DEET

- We used Sinz/Küchlin solver that can handle non-CNF formulae easily.
- It took the solver a fraction of a second to find the counterexample.
- We tried it on an example with 2000 statements and 6000 variables. It took the solver less than 2 seconds to find two counterexamples on a 1.2MHz Athlon PC .

Status and Future Directions

- Our thesis: DEET can be an efficient and cost-effective prelude to more exhaustive testing or verification.

- Its scalability and utility for error detection needs to be shown through practical experimentation.