Performance Specifications
Based upon Complete Profiles

Joan Krone
William F. Ogden
Murali Sitaraman
Our Starting Point

D. Parnas:
• A good specification should tell a client everything he needs to know about a component and nothing more.

Us:
• A client needs to know not only about the functionality provided by a component, but also about its performance.
Goals for a Performance Specification Mechanism

It should support:

• Abstracting away confusing details
• Retaining adequate precision (completeness)
• Scaling for arbitrarily large components
• Verifying correctness of compositions
• Extending functional specifications

and critically:

• Describing commonalities
Commonality Identification
Example

Various “Sorting” Implementations
Abstract Sorting Component (Prioritizer)

Type_Fam EntryKeeper
Oper Add_Entry
Oper Change_Modes
Oper Remove_a_Smallest

Functionality Abstraction

bubble quick select heap tree ... insert
Abstract Sorting Component (Prioritizer)

Type_Fam EntryKeeper
Oper Add_Entry
Oper Change_Modes
Oper Remove_a_Smallest

Functionality Abstraction

Quad_Ch_Md (PCQC)
LoG_Rmv (PCGR)
Linear_Add_E (PCLA)

Performance Abstraction

bubble quick select heap tree insert

...
Simple Profile Example with Stacks

Concept Stack_Template( type Entry; eval Max_Depth: 

::

Type_Family Stack ⊆ Str(Entry);

::

Operation Push( alters E: Entry; updates S: Stack );

requires |S| < Max_Depth;

ensures S = ⟨#E⟩•#S;

Operation Pop( replaces R: Entry; updates S: Stack );

requires |S| > 0;

ensures #S = ⟨R⟩•S;

Operation Depth_of( preserves S: Stack ): Integer;

ensures Depth_of = ( |S| );

::
**Enhancement** Flipping\_Capability for Stack\_Template;

**Operation** Flip( updates S: Stack );

ensures S = S^{Rev};

end Flipping\_Capability;

Possible Implementation:

**Realization** Obvious\_F\_C\_Realiz for Flipping\_Capability

**Procedure** Flip( updates S: Stack );

Var Next\_Entry: Entry;
Var S\_Flipped: Stack;
While Depth\_of( S ) \neq 0

affecting S, S\_Flipped, Next\_Entry;

maintaining \#S = S\_Flipped^{Rev} \circ S and

Entry.Is\_Init(Next\_Entry);

decreasing \mid S\mid;

do

Pop( Next\_Entry, S );
Push( Next\_Entry, S\_Flipped );

end;
S := S\_Flipped;
end Flip;
end Obvious\_F\_C\_Realiz;
An Example Profile

Profile SSSF short_for Stack_Space_Conscious_Flip for Flipping_Capability for Stack_Template with_profile SSC;
Defines SSSF\_F1, SSSF\_F2: \(\mathbb{R} \geq 0\);
Defines SSSF\_FMC1, SSSF\_FMC2: \(\mathbb{N}\);
Operation Flip( updates S: Stack );
  duration SSSF\_F1 + Entry.I\_Dur + Stack.I\_Dur + 
  Entry.F\_IV\_Dur + Stack.F\_IV\_Dur +
  (SSSF\_F2 + Entry.I\_Dur + Entry.F\_IV\_Dur) \cdot |S|;
manip\_disp (SSSF\_FMC1 + Entry.I\_Disp + Stack.I\_Disp) +
  Max( SSSF\_FMC2, Entry.IM\_Disp, Entry.F\_IVM\_Disp );
end SSSF;
\[ \text{duration } SSCF_{F_1} + \text{Entry.I.Dur} + \text{Stack.I.Dur} + \text{Entry.F.IV.Dur} + \]  
\[ \text{Stack.F.IV.Dur} + (SSCF_{F_2} + \text{Entry.I.Dur} + \text{Entry.F.IV.Dur}) \cdot |#S|; \]

**Realization**  Obvious F.C Realiz for Flipping Capability

**Definition**  
\[ SSCF_{F_1} : \mathbb{R}^{\geq 0} = (\text{Dur}_{\text{Call}}(1) + \text{SCDp} + \text{Int.Dur}_{\neq} + \text{Dur} := ) ; \]

**Definition**  
\[ SSCF_{F_2} : \mathbb{R}^{\geq 0} = (\text{SCDp} + \text{Int.Dur}_{\neq} + \text{SCPo}_1 + \text{SSCP}_u); \]

**Definition**  
\[ SSCF_{FMC_1} : \mathbb{N} = \ldots \]

**Procedure**  \( \text{Flip( updates S: Stack )} ; \)

\[ \begin{align*} 
\text{Var Next_Entry: Entry; } \\
\text{Var S_Flipped: Stack; } \\
\text{While } \text{Depth.of( S )} \neq 0 \\
\text{affecting S, S_Flipped, Next_Entry; } \\
\text{maintaining } #S = \text{S_Flipped}^{\text{Rev}} \circ S \text{ and Entry.Is_Init(Next_Entry); } \\
\text{decreasing } |S| ; \\
\text{elapsed_time ( SSCF}_{F_2} + \text{Entry.I.Dur} + \text{Entry.F.IV.Dur}) \cdot |S_{\text{Flipped}}| ; \\
\text{do } \\
\text{Pop( Next_Entry, S ); } \\
\text{Push( Next_Entry, S_Flipped ); } \\
\text{end; } \\
S := S_{\text{Flipped}} ; \\
\text{end Flip; } 
\end{align*} \]
Profile SSC short_for Space_Conscious for Stack_Template;

  Defines  SSC_l, SSC_l1, SSC_F, SSC_Po1, SSC_Pu, SSC_C, SSC_C1, SSC_Dp, SSC_RC: \( \mathbb{R} \geq 0 \);

  Type_Family Stack;

    Initialization
      duration  SSC_l + (SSC_l1 + Entry.I_Dur) \cdot \text{Max_Depth};

    Operation Pop( replaces R: Entry; updates S: Stack );
      duration  SSC_Po1 + Entry.I_Dur + Entry.F_Dur(#R);\(^\dagger\)

    Operation Push( alters E: Entry; updates S: Stack );
      ensures  Entry.Is_Init(E);\(^\ddagger\)
      duration  SSC_Pu;

    Operation Depth_of( preserves S: Stack ): Integer;
      duration  SSC_Dp;

  end SSC;

\(^\dagger\)Note that this duration expression is split between the externally defined terms for the duration of an Entry initialization, Entry.I_Dur, and the finalization of the incoming value of R, Entry.F_Dur(#R) and the internally defined term SSC_Po1.

\(^\ddagger\)Note that this extension of the functional specification for Push is essential for achieving tight performance specifications.
So, the **Profile** construct is a performance specification mechanism that supports:

- Abstracting away confusing details
- Retaining adequate precision (completeness)
- Scaling for arbitrarily large components
- Verifying correctness of compositions
- Extending functional specifications
- Describing commonalities

It (or something quite similar) should be included in any serious language for component specification and verification.
What Else is There?

• What happens with displacement (space).
• How component composition works.
  – Multiple profiles for a constituent.
• How large components can have simple profiles.
• How to formally verify profiles.
• When to supplement an object model.