Automatic Re-Coding of Reference Code into Structured and Analyzable Models

Pramod Chandraiah and Rainer Doemer

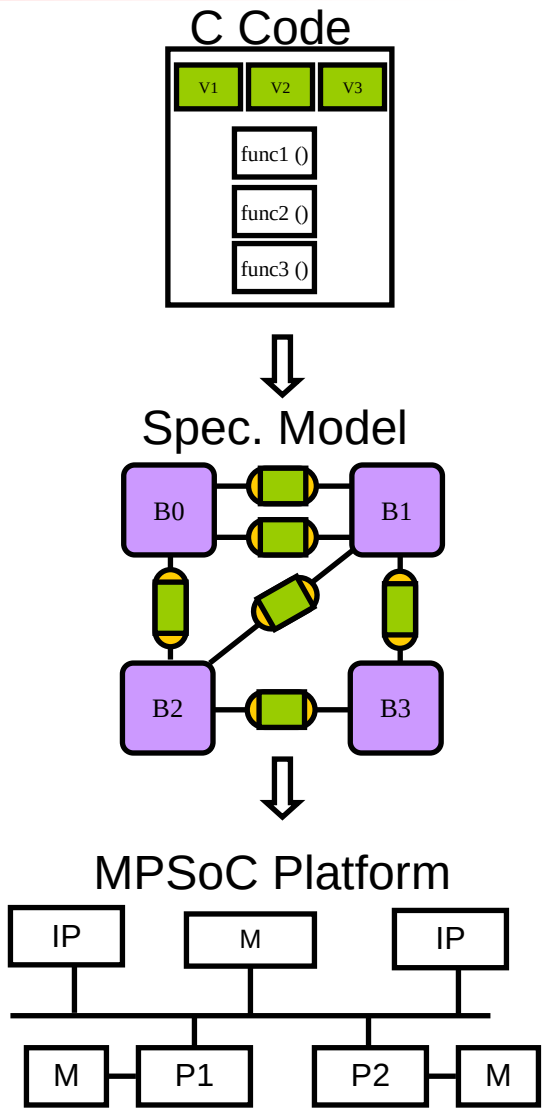
Center for Embedded Computer Systems

University of California, Irvine

UCIrvine
University of California, Irvine

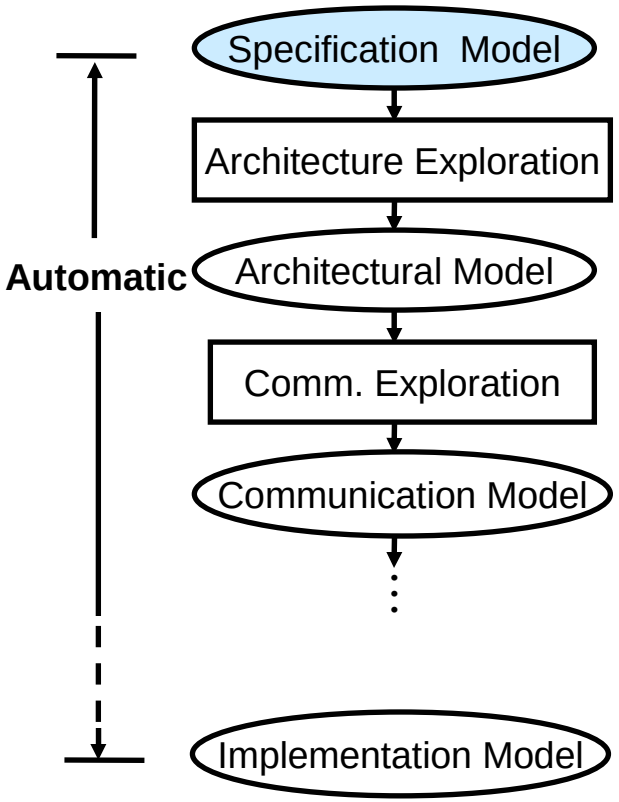
Introduction

- System level design
  - Specification to implementation
  - Needs system model
- Specification model
  - Starting point of the design
  - Structured and Analyzable model
    - Explicit computation & communication
    - Analyzable, synthesizable, verifiable
- C reference code
  - Actual starting point of design
  - Ambiguous
    - Non-definite computation & communication
    - Insufficient for Architectural Exploration
- Re-Coding C code to specification model
  - Creating structural hierarchy
  - Creating Analyzable interface for behavioral blocks



Motivation

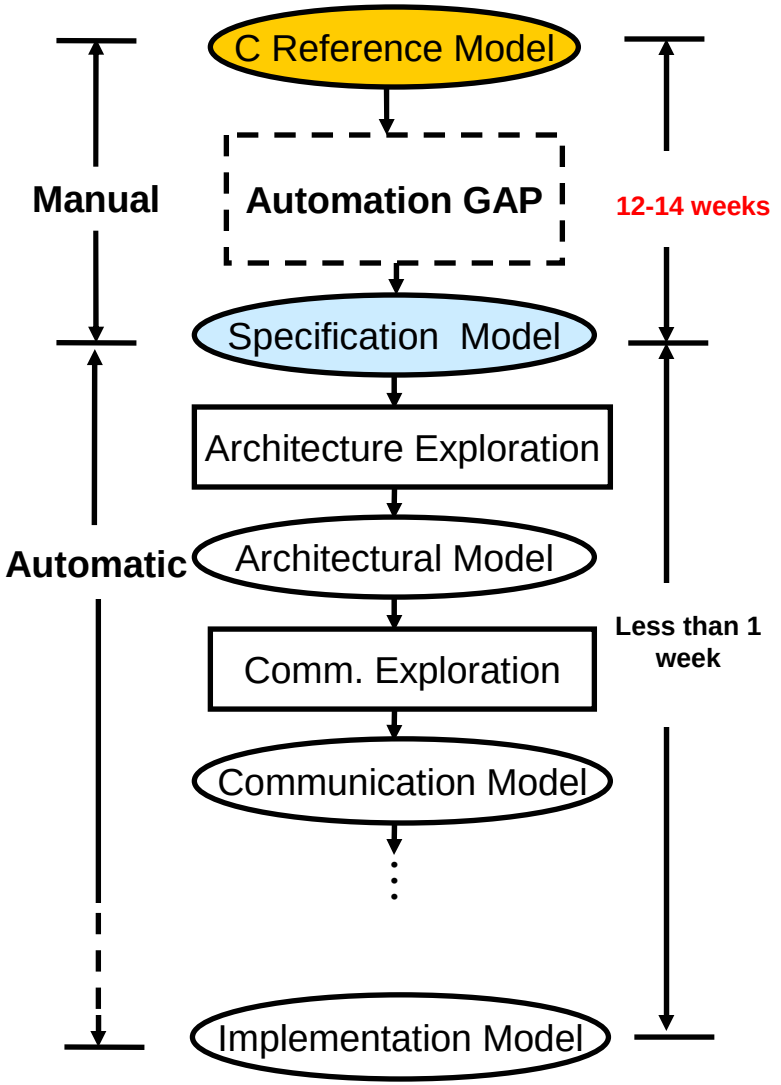
- SpecC design methodology
  - Starts from Specification Model
  - Architectural exploration
  - Communication exploration
  - …
- Architectural exploration
  - Code partitioning
  - Grouping/Re-grouping behaviors
  - Mapping behaviors to PEs
- Specification Model
  - Behaviors are the basic unit of computation
  - All the explorations (partitioning and mapping) happen at the granularity specified by the behaviors



Source: ***System Design: A practical guide with SpecC***

Motivation

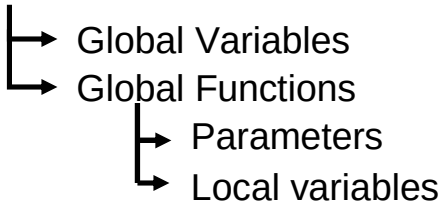
- Absence of structural hierarchy limit the effectiveness of system design tools
  - Architecture exploration tools
    - Code partitioning and mapping
    - Analyzability
  - System level performance analysis
- Manual re-coding of C into SoC model in SLDL is time consuming
- Proposed Solution: Automatic behavior creation
  - Enables automatic architectural exploration



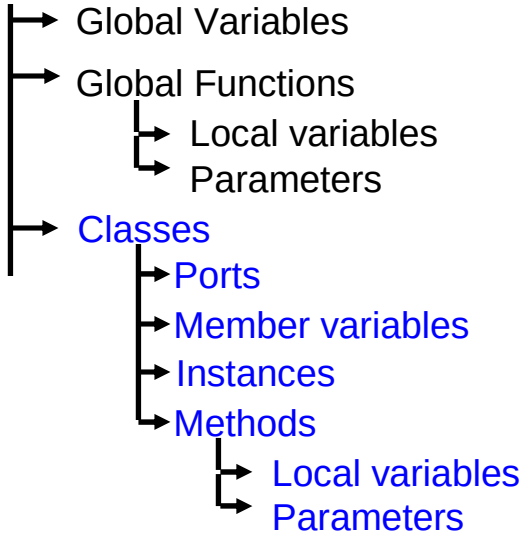
Source: ***System Design: A practical guide with SpecC***

Advantages of behaviors

- Explicit interface
  - Ports
    - Name, type and direction
  - Port map
    - Mapping known at compile time
- Additional level of Scope
  - C has only 2 levels
    - Global
    - Local
  - Behaviors in SLDL provide 3 levels
    - Global
    - Local
    - Class



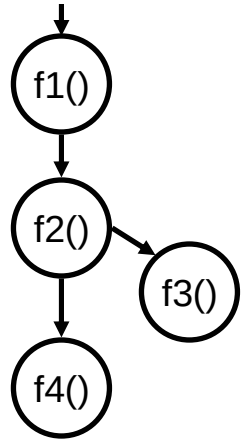
Syntactical hierarchy in C code



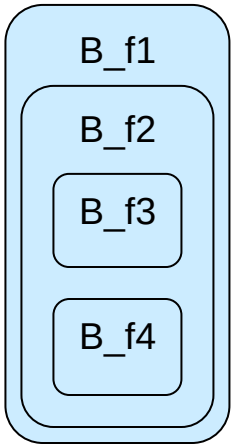
Syntactical hierarchy in SLDL code

Creating Structural Hierarchy

- Use the partial structure available in the form of functions
- Encapsulating C code blocks
  - Determining the statically analyzable interface
  - Re-coding to encapsulate the code in a behavior
  - Instantiation of behavior



Function call hierarchy



Structural hierarchy

Analyzable Interface

- Accumulated accesses to variable in the code block
  - In case of functions, function parameters, return variables
  - In case of statements, variables in the code block, excluding the local variables within the block
- Cumulative Access Types (CATs)
  - Read *(a, i, b[10])*
  - Write *(c)*
  - Read-Write *(s)*
- Safe assumption made about arrays
  - As in case of array *b* as the value of *i is* not statically known

```
int func( int w, int x, int *p)
{ *p  = w+x+*p}

pointer = &s
c= func (a, b[i], pointer)
```

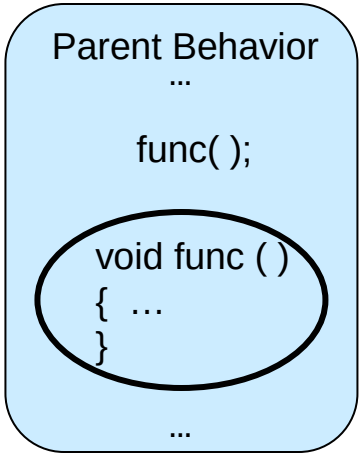
Code Snippet

I: {R: a, R: b[10], R: i, RW: s, W: c }

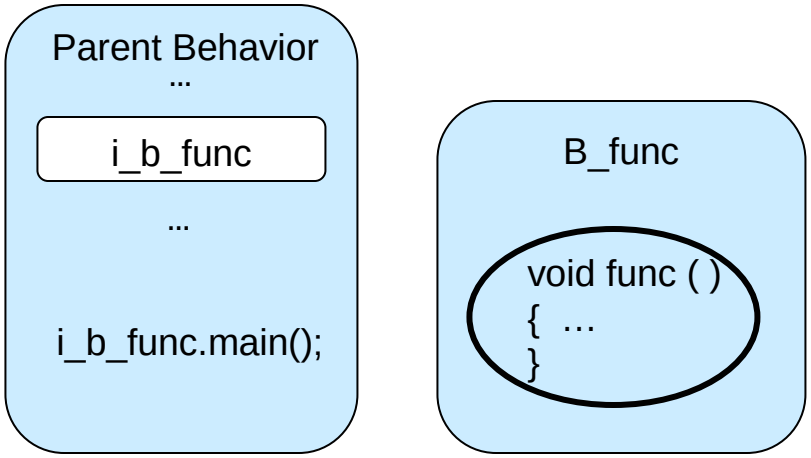
Interface of *func()*

Encapsulation and Instantiation

- Function encapsulation
  - Create port list from the interface
    - *Read* access results in *in* port
    - *Write* access results in *out* port
    - *RW* access results in *inout* port
  - Return variable becomes an *out* port
  - Create behavior body
- Behavior Instantiation
  - Create port map
    - From function arguments
    - Return value
  - Replace function call with instance call



Initial structure with global function



Global function encapsulated in behavior (syntactical structure)

Re-coding Complexities

- Variables used in the port map must be made available in the class scope
  - Example, *a, b,i1, s*
- Need to handle the difference in semantics of functions and behaviors
  - Especially, Call-by-value semantics
  - Call-by-value parameter can only be an *in* port

```
behavior B ( in int p1, in int p2,  out int result)
{
    void main( )
    {  int i1, a, b[10], s, *pa;
       pa = &s;
       …..
       result = f1(a, b[i1], pa);
       …..
    }
    int f1( int w, int x, int *p)
    { *p  = w+x+*p;
       return *p;
    }
};
```

Initial code with function *f 1()*

```
behavior B (in int p1, in int p2, out int  result) {

   int a, b[10], i1, s;

   I_B_f1(a, b, i1, s, result);
   void main( )
   {
       int *pa;
       a = p1+p2;
       s = p1-p2;
       pa = &s;
       …..
      I_ B_f1.main();
       …..
   }
} ;
```

f1() replaced with instance call

Re-coding Complexities

- Implicit function return value
  - Example, function call in the conditional expression of an *if* statement
  - Explicit return value is introduced
- Evaluation of expressions in function arguments
  - Example, *w+x* must be either evaluated first or inside the behavior

```
1. int func (int, int, int);
2. /*…*/
3. if (func (w+x, y, z))
4. { /* …*/
5. }
```

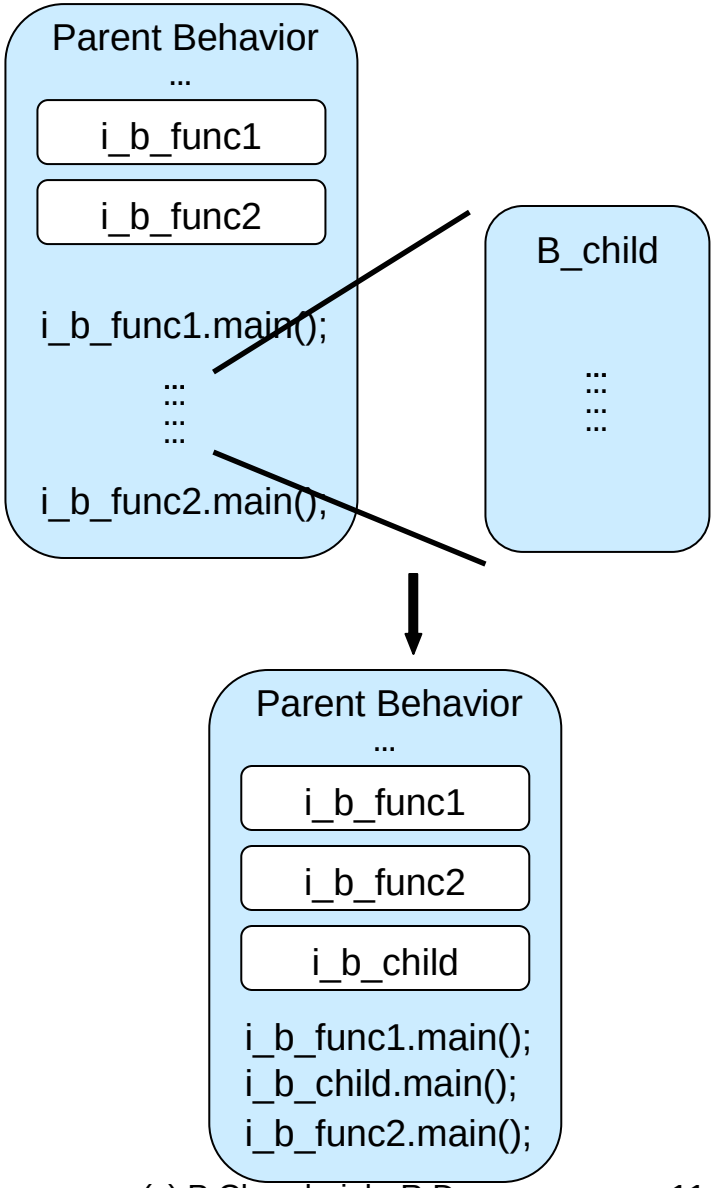
Initial code with function *func()*

```
1. behavior B_func (in int, in int, in int, out int);
2. /*…*/
3. int wx, retval;
4. B_func I_B_func (wx, y, z, retval); //Instance
5. /*…*/
6. wx = w+x;
7. I_B_func.main();
8. if (retval)
9. { /* …*/
10.}
```

Code after replacing *func()* with behavior

Encapsulating Statements

- Encapsulating functions works on the function calls
- The statements that exist between function calls must be encapsulated, as well



Establishing connectivity

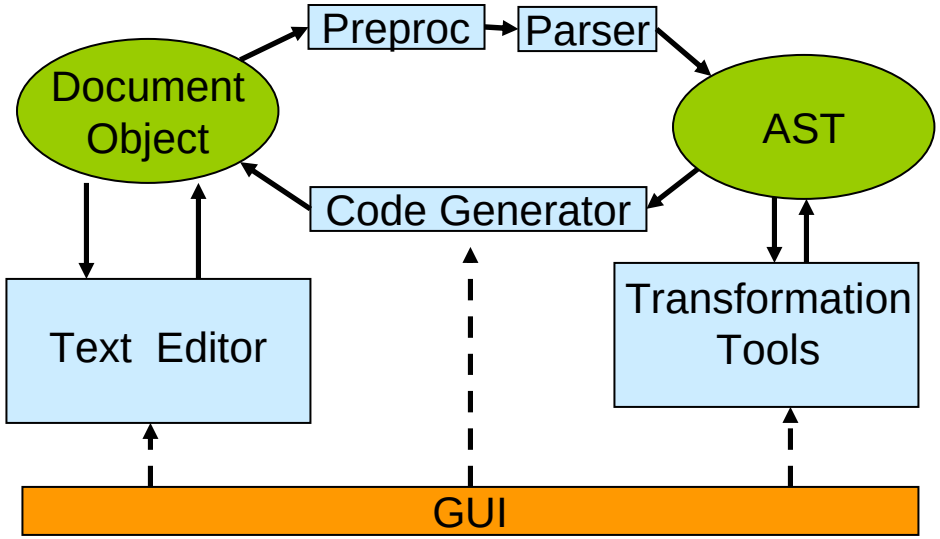
- After encapsulating functions, global variables must be localized to the behavioral partitions
- Procedure
  - Finding the global variable
  - Determining the functions and behaviors accessing it
  - Migrate it to the lowest most common parent behavior
  - Providing access to the new localized variables by inserting ports

Restrictions

- Encapsulating functions applies only to internal functions
- Encapsulating statements not allowed in presence of *goto* statements

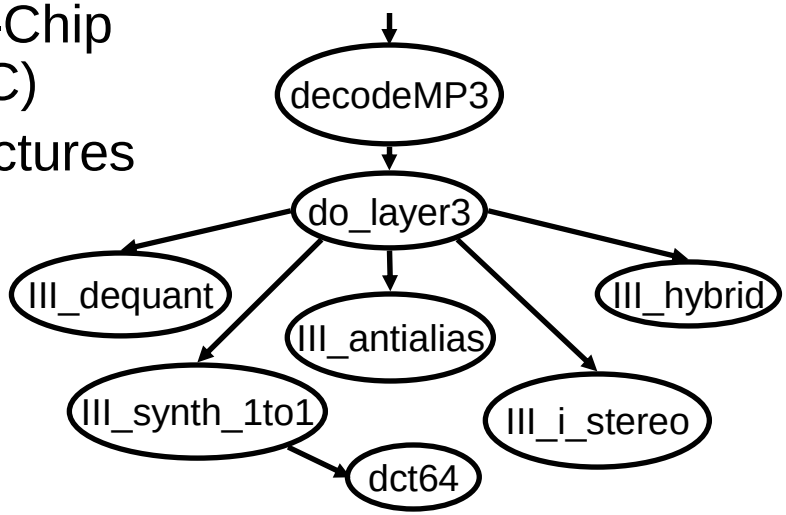
Interactive Source Re-coder

- Controlled interactive approach
- Automatic programming
- It's a union of
  - Textual Editor
  - Abstract Syntax Tree
  - Preprocessor/Parser
  - Transformation and analysis functions
  - Code generator
- Main Characteristics
  - Computes points-to information with a click
  - Re-codes pointers with a click
  - Applies Source-to-Source transformations on-the-fly

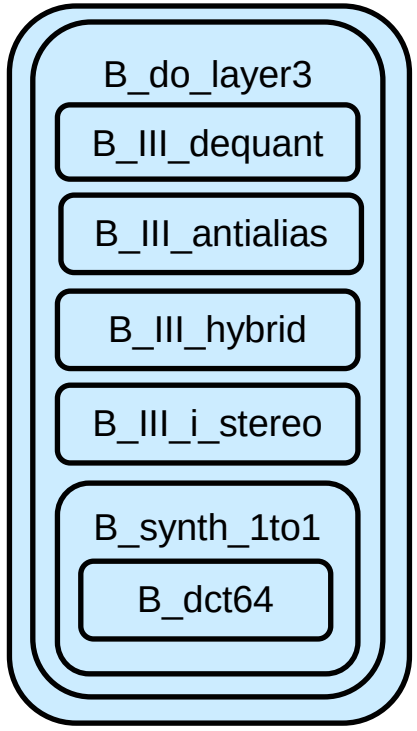


Experiments

- Specification model of MP3 Decoder
  - Input C code had 30 functions
  - 43 behaviors were introduced
- Design Explorations
  - Using System-on-Chip Environment (SEC)
  - 6 different architectures



Partial function hierarchy in MP3 code



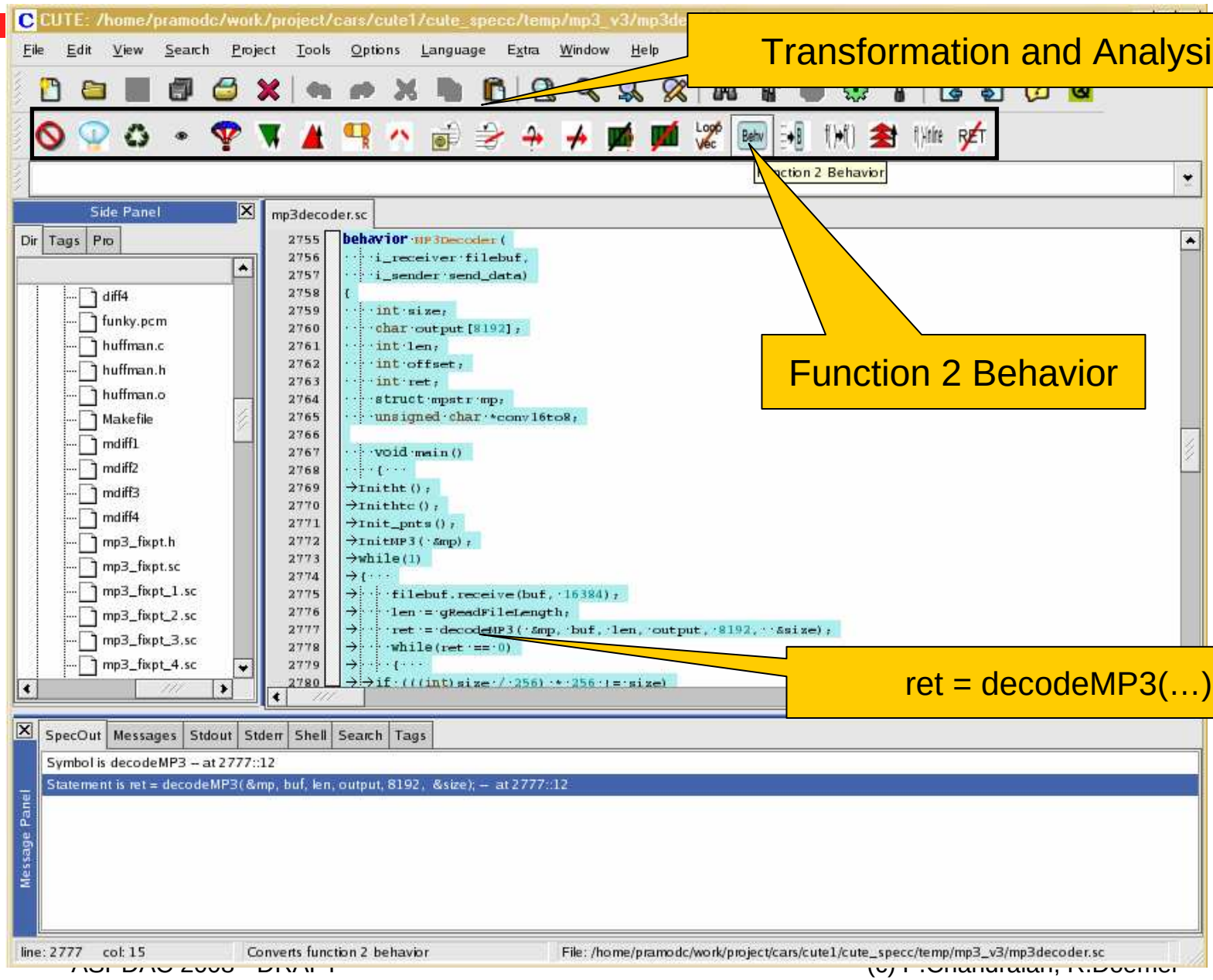
Structural hierarchy

Gains

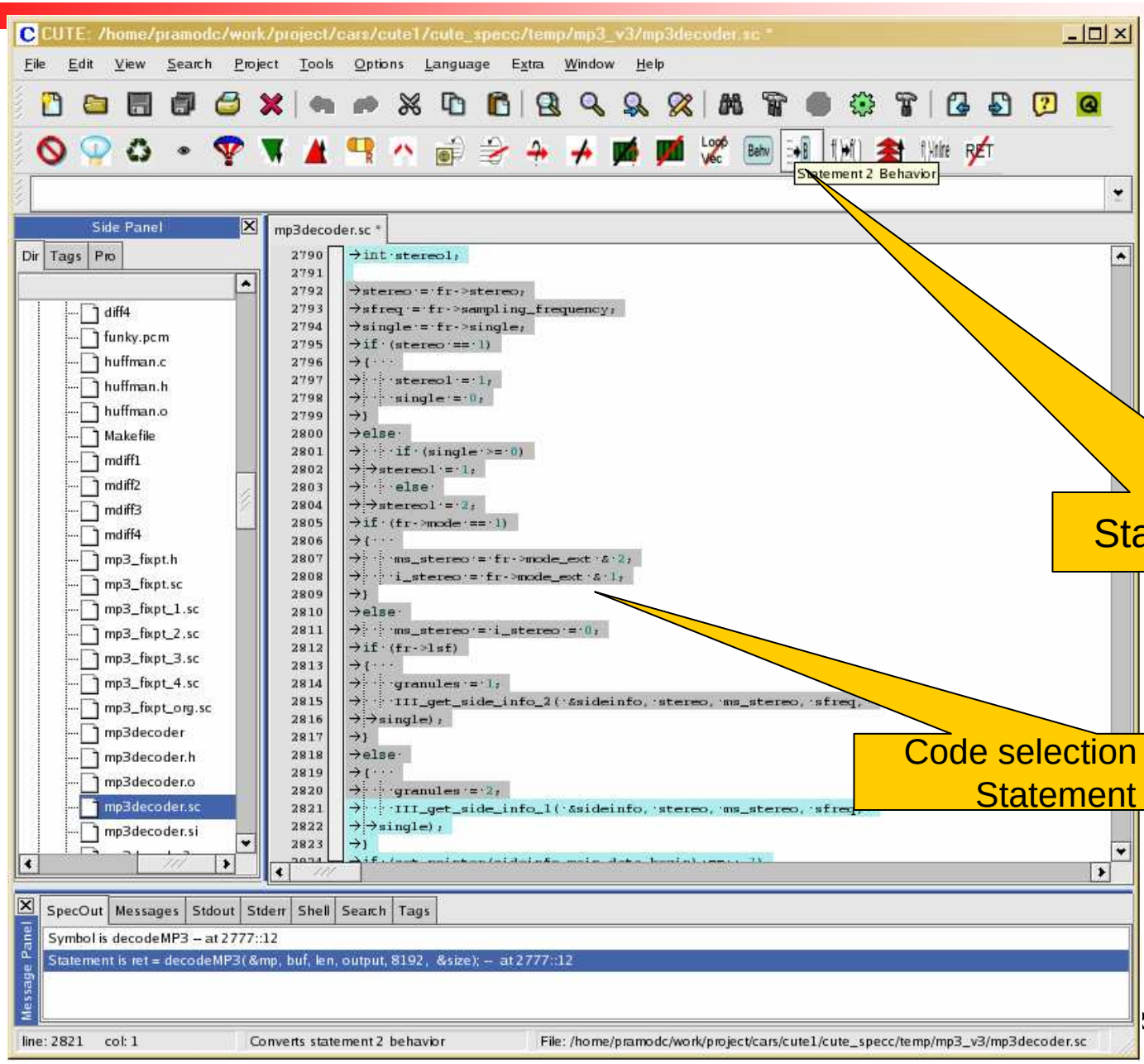
- Design examples
  - JPEG, MP3, GSM
- Introduced behaviors
  - Manual
  - Source Re-Coder
- Manual time
  - Created manually by different designers
- Re-coding time
  - Time to implement the transformations using Source Re-coder
  - In the order of minutes
- Significant productivity gains

Properties	JPEG	Float-MP3	Fix-MP3	GSM
Lines of C code	1K	3K	10K	10K
C Functions	32	30	67	163
Lines of SpecC code	1.6K	7K	13K	7K
Behaviors created	28	43	54	70
Re-Coding time	≈ 30 mins	≈ 35 mins	≈ 40 mins	≈ 50 mins
Manual time	1.5 weeks	3 weeks	2 weeks	4 weeks
Productivity gain	120	205	120	192

Source Re-coder



Source Re-coder



Conclusions

- SoC specification
  - Structured, Analyzable, and synthesizable
  - C Reference Models act as starting point
- C code to Structured SoC specification
  - Time-consuming and error-prone re-coding
- Creating structural hierarchy
  - Automatic creation of behaviors from C functions and statements
  - Function-call chain transformed to behavioral hierarchy
- Interactive Source Re-coder
  - Designer-in-the-loop approach to recode C into SoC specification
  - Programming using source-level transformations
  - Integrates automatic behavior creation
- Compared to manual programming, source re-coder results in significant productivity gains
- Facilitates architectural exploration by enabling easy partitioning of code
- Enables faster path to the implementation

References

- S. Abdi, J. Peng, H. Yu, D. Shin, A. Gerstlauer, R. D¨omer,and D. Gajski. System-on-chip environment (SCE version 2.2.0 beta): Tutorial. Technical Report CECS-TR-03-41, CECS, University of California, Irvine, 2003
- F. Ghenassia. *Transaction-Level Modeling with SystemC : TLM Concepts and Applications for Embedded Systems*. Springer- Verlag, 2006.
- Sprint parallelizes real life applications for embedded systems. http://www.imec.be/design/sprint/.
- A. Pimentel, L.O.Hertzberger, P. Lieverse, and P. Wolf. Exploring embedded-systems architectures with artemis. *IEEE Transactions on Computers*, 34(1), November 2001.
- A. Jerraya, H. Tenhunen, and W. Wolf. Guest editors' introduction: Multiprocessor systems-on-chips. *Computer*, 38(7):36.40, 2005
- A. Gerstlauer, R. D¨omer, J. Peng, and D. D. Gajski. *System Design: A Practical Guide with SpecC*. Kluwer Academic Publishers, 2001.
- P. Chandraiah and R. D¨omer. Specication and design of an mp3 audio decoder. Technical Report CECS-TR-05-04, CECS, University of California, Irvine, 2005.