Automatic Re-coding of Reference Code into Structured and Analyzable SoC Models

Pramod Chandraiah
Center for Embedded Computer Systems
University of California, Irvine
pramodc@cecs.uci.edu

Rainer Dömer
Center for Embedded Computer Systems
University of California, Irvine
doemer@cecs.uci.edu

Abstract— The quality of the input system model has a direct bearing on the effectiveness of the system exploration and synthesis tools. Given a well-structured system model, tools today are effective in generating efficient implementations. However, readily available reference C codes are not conducive for system synthesis as they lack the necessary structure and analyzability needed by the design flow. Usually reference C code is manually converted into a SoC model by applying necessary transformations. The type of transformations depends on the underlying design flow and tools. Proper structural hierarchy is one essential feature needed for architectural exploration. In this paper, we provide automatic C code transformations to encapsulate functions and insert structural hierarchy to create well-structured and analyzable SoC models. Our automatic transformations, combined with interactive application of the designer's knowledge and experience, enable faster creation of structural hierarchy in C models and hence result in significant reduction of the overall design time.

Fig. 1. Motivation: Design time of MP3 decoder in a refinement-based system design flow [1].

I. INTRODUCTION

The large availability of embedded applications in the form of reference C code has made C implementations a natural starting point of most System-On-Chip (SoC) design processes. Easily obtained from open-source projects and standardizing committees, these C codes not only serve as reference models for functional verification purposes, but also can be used for deriving end SoC implementations. However, there are significant obstacles in directly using such C sources for the SoC design process. First of all, these C codes typically come from different sources, such as the signal-processing or the general-purpose programming community. Often, these sources are designed and optimized to run on a regular PC environment where the intended target machine architecture is known. However, such code is usually not suitable for SoC design where a custom target architecture with multiple processing elements is still to be derived, and where a chain of tools (rather than a single compiler) will be used to derive the end implementation. Second, the C language itself poses numerous challenges for system design tools. The freedom available through a range of programming constructs make the models ambiguous to the system design tools. Commonly used constructs such as C pointers, dynamic memory allocation, recursion, and more, result in ambiguities and negatively affect the analyzability, verifiability and synthesize-ability of the tools.

System design tools require a clear structure and analyzability in their input models. The model features expected by the de-
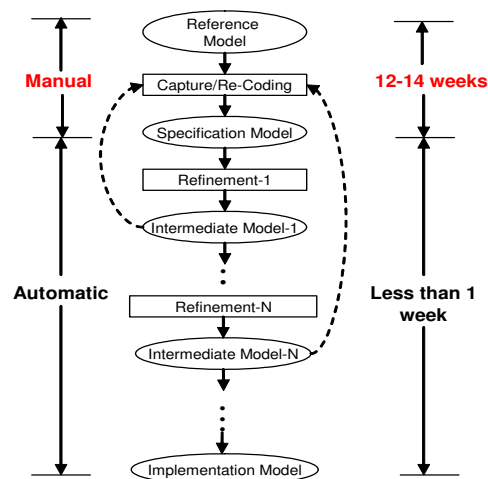
sign tools depend on the tasks undertaken by the tools. For instance, system-level architecture exploration, which performs tasks such as partitioning and mapping of application onto an abstract architectural platform, often requires explicit specification of code and data partitions in models. Similarly, synthesis tools typically require models without pointers, recursive functions, dynamic memory allocation, and so on. Since structural hierarchy is one essential feature needed by system design tools, we address the problem of creating well-structured SoC models in this paper.

A. Motivation

Structural hierarchy is a critical property needed by system design tools [7]. With a well-structured model, the architecture exploration tools can attempt different code partitioning by grouping and re-grouping different encapsulated blocks and mapping them onto different components in the virtual architecture.

In order to study the intricacies and complications involved in creating a system specification, we applied a top-down design methodology, as shown in Figure 1, to the example of a multimedia application, a MP3 audio decoder. Here, the design process starts with an abstract specification model which is then refined to create models at lower abstraction levels, including transaction-level, bus-functional and implementation models. After a series of refinement steps, an actual implementation model is finally derived. Each of the refinement steps in the design flow is automated to the extent that model generation is
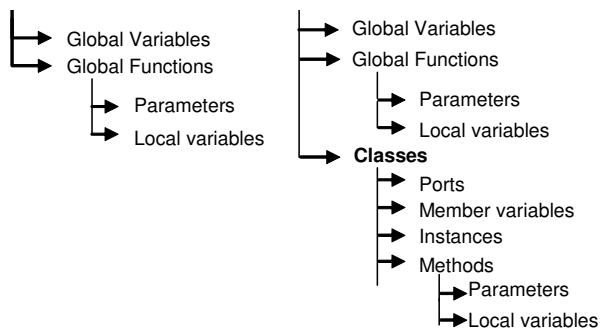
(a) Syntactical hierarchy in C code   (b) Syntactical hierarchy in SLDL code

Fig. 2. Hierarchy of scopes in C and SLDL

Fig. 3. Introducing structural hierarchy

fully automatic, and the designer has to only make the design decisions such as component allocation, mapping and scheduling. Due to this automation, we were able to implement our MP3 decoder model, an industry-size application, in less than one week [3]. In contrast, manually re-coding the MP3 reference code into a structured specification model took 12-14 weeks. Writing and re-writing this model was the main bottleneck of the whole design process. More than 90% of the overall design time was spent in creating this model.

Also, we need to emphasize that specification capturing is not a one time task. Every time a change in the design is required for a successful refinement step, it is necessary to re-code/change the input specification (as shown by back arrows in Figure 1), making the whole task of specification writing iterative. Such interruptions in the design flow cause costly delays. The problem of lengthy re-coding of models is also emphasized in [11, 8].

In this paper we present a set of automatic source code transformations needed to create structural hierarchy in the system model. These transformations are integrated into a programming environment which facilitates fast conversion of "flat" C code into a hierarchical system model.

B. Related Work

The effectiveness of today's system design flows depends on the quality of the initial specification model. However, the creation of this model from the readily available C references is a problem which has not received much attention. Typically, the designer is expected to manually create the specification of acceptable quality to suite tools. [12] provides user guided transformations for functional partitioning and structural reorganization to transform a system level specification in system design language (SDL [10]) into a HW/SW architecture in C/VHDL. Unlike this work, our work focuses on creating a model with structural hierarchy from a flat unstructured C code. The *Compaan* tool-set [13] transforms a sequential application written in Matlab into a Kahn Process Network that acts as input model for architecture exploration of multiprocessor architectures. *Sprint* [14] from IMEC transforms a sequential C program to a task-level pipelined program in SystemC, with user-defined task boundaries. Unlike these, the primary focus of our work is to create a well-structured and analyzable model in System Level Design Language (SLDL) from C code under the control of the designer. In our approach, the designer is not restricted to one type of model. Instead, she/he has complete
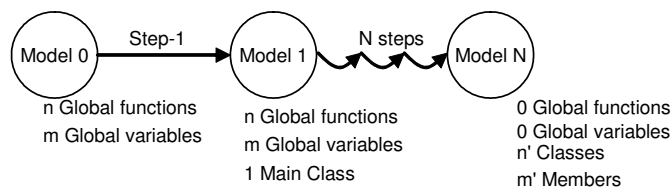
control ("designer-in-the-loop") to code/re-code the model in order to arrive at the most suitable specification model.

II. MODELING STRUCTURAL HIERARCHY

The "flat" C language is insufficient when it comes to modeling structural hierarchy. System Level Design Languages (SLDLs) such as SystemC [9] and SpecC [7], which are supersets of the C language, have the necessary language extensions to specify structural hierarchy and to isolate computation from communication. In particular, SLDLs provide an extra level of scope compared to the C language. This difference is depicted in Figure 2. In case of C, there are only 2 major levels of hierarchy, *global scope* which consists of variables and global functions, and *local scope* consisting of function parameters and local variables. Elements in the global scope are globally available across the whole program.

In SLDLs, in contrast, there is an additional level of hierarchy available through classes which represent modules/behaviors and channels. This *class scope* contains ports, member variables, instances of other classes, and methods. Methods, just like functions, have their own local scope consisting of local variables and parameters [1]. Connectivity from higher levels to the class scope is available through ports. This additional level of hierarchy available in SLDLs is used to describe structural hierarchy in SoC models.

Though C functions can be thought of as computation encapsulation entities, as such they are insufficient in providing isolation from communication. System design tools require computation blocks, communication channels and interfaces to be explicitly specified. SLDLs provide behaviors and channels to capture the computation and communication, respectively. These encapsulation entities provide a means to explicitly and unambiguously specify the communication interface through ports. Ports provide a means to specify not only the data type, but also the type of access (Read, Write, Read-Write) using *in*, *out*, and *inout* directions. Use of behaviors/modules with proper port mappings makes it possible to statically represent and analyze the structural hierarchy and connectivity of a design model.

A. Problem Definition

Introducing structural hierarchy in flat C code in order to create a well-structured SoC model in SLDL involves introducing this new class scope and encapsulating the global variables and global functions. Local scopes of functions are also encapsulated within the class scope. This problem of introducing structural hierarchy is depicted in Figure 3. The initial flat C model,

---

[1]Minor levels of scopes are also possible through compound statements in both C and SLDLs. These, however, omitted in the figure for brevity.
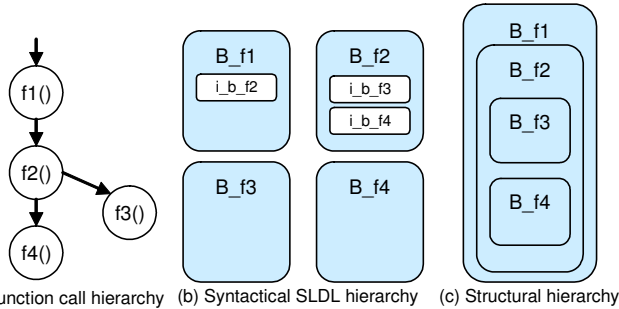
(a) Function call hierarchy   (b) Syntactical SLDL hierarchy   (c) Structural hierarchy

Fig. 4. Converting function call chain to structural hierarchy

```
int func( int w, int x, int *p)
{ *p = w+x+*p}
/* ... */
pointer = &s
c= func (a, b[i], pointer)
/*...*/
}
```

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

Complete interface

(a) Code Snippet   (b) Interface of *func()*

Fig. 5. Determining statically analyzable interface of code blocks

```
1. int a[32], b[16], c, d, x;
2. …
3. x = i *i;        //CAT(x) in this block is RW
4. a[i]++;          //CAT(a) in this block is RW
5. a[2i] = c+d;     //CAT(c,d)  in this block is R
6. b[i] = c*d-x;    //CAT(b)  in this block is W
```

Fig. 6. Cumulative access of variables in a block

Model0, with n global variables and m global functions, is first converted into a SLDL model, Model1, with one initial Main class. Typically, this initial class becomes the testbench of the design into which other classes including the design-under-test (DUT) are successively instantiated. Following this, the global functions in the model are encapsulated in a new class scope in a series of N iterative steps creating n' classes with m' member variables, and ideally 0 global entities.

This problem of introducing structural hierarchy includes 3 sub-problems:

1. Encapsulating n global functions into n' classes, by following the function call hierarchy in top-down order

2. Analysis to determine the variables affected by introduction of new class scopes

3. Migration of global and local variables into appropriate class scope, global scope or local scope.

4. Establishing connections through channels, ports and parameters to make the communication explicit.

As we can see, creating structural hierarchy is a complex process, which is very error-prone and time consuming when performed manually. In this paper, we propose a technique that automates most of these tasks. In particular, the tasks of analysis, encapsulating and establishing connections can be automated. Only the tasks requiring decision making based on application knowledge and designer experience, such as determining the functions to be encapsulated and the destination of migrating variables, are controlled by the designer. In the next section, we will present our source level transformations that implement this approach.

III. CREATING STRUCTURAL HIERARCHY

The partial structure available in the C code in the form of functions can be used as starting point to create a proper structural hierarchy in the model. Most functions can be encapsulated into separate behaviors to create a modular SoC model. By static analysis, the function call hierarchy of the overall program is generated. Figure 4(a) shows an example function call hierarchy[2]. Function *f1()* calls *f2()* which in turn calls *f3()* and *f4()*. This call chain is then traversed in the hierarchical order and the functions are encapsulated in behavior shells with definite interface. Figure 4(b) shows the resulting syntactical structure after encapsulation. Figure 4(c) shows the structural

[2]Note that function hierarchy only provides information about a function and the child functions it calls. It does not indicate the order of the functions.

hierarchy of the resulting model.

More specifically, the process of encapsulating C code blocks into behaviors involves multiple steps:

(a) Determining the statically analyzable interface of the selected block of C code

(b) Re-coding to encapsulate the block in a behavior/module class

(c) Instantiating the new class and replacing the function call with a call to the new instance

A. Statically analyzable interface

The interface of an encapsulating class is the list of data items the class accesses. An unambiguous interface contains access type information (direction) and does not include pointers, and does not depend on run-time values. When all the classes in the model have a well-defined interface, the design tools can fully rely on this interface without having to analyze the body of the block. Figure 5 shows a piece of C code and the corresponding interface of a function *func()*.

The complete interface contains the access type information (such as read/write). This information will be later used to generate the appropriate direction of the ports (*in, out, inout*). The interface of a block is determined by analyzing the Cumulative Access Types (CAT) of all the variables within the block, and reveals the access type of all contained scalar and vector variables. The overall access of a variable in a block is the accumulation of all local accesses in the individual expressions, as shown in Figure 6[4]. This can be represented as

$$CAT(var) = \cup Access(expr), \forall expr \in block$$

We classify the cumulative accesses to variables into 3 categories, Read(R), Write(W), and Read-Write(RW). Figure 6 shows variables with 3 different cumulative access types (CATs). Since we want the interface to be statically analyzable, for any vector accessed using a non-constant index variable, a safe assumption is made and the whole array is assumed to be accessed. If the access to the specific array elements cannot be determined statically, as for *b[i]* in Figure 5, the complete interface includes the whole array *b* and the index *i*. If there is a

(a) Initial structure with global function   (b) Global function encapsulated in behavior (syntactical structure)

Fig. 7. Encapsulating function into behavior

```
1.  behavior B ( in int p1, in int p2,       1.  behavior  B_f1( in int w, in  int
    out int result)                              x[10], in int i, inout int s, out int c) {
2.  {                                         2.     void main()
3.     void main( )                           3.     { c= f1(w, x[i], &s);
4.     {                                       4.     }
5.        int i1, a, b[10], s, *pa;           5.     int f1( int w, int x, int *p)
6.        a = p1+p2;                           6.     { *p  = w+x+*p;
7.        s = p1-p2;                           7.         return *p;
8.        pa = &s;                             8.     }
9.        .....                                9. };
10.       result = f1(a, b[i1], pa);          10.  behavior B (in int p1, in int p2, out int
11.       .....                                    result) {
12.    }                                       11.     int a, b[10], i1, s;
13.    int f1( int w, int x, int *p)           12.     //Instantiate child behavior here
14.    { *p  = w+x+*p;                         13.     I_B_f1(a, b, i1, s, result);
15.        return *p;                          14.     void main( )
16.    }                                       15.     {
17.};                                          16.         int *pa;
                                               17.         a = p1+p2;
                                               18.         s = p1-p2;
                                               19.         pa = &s;
                                               20.         .....
                                               21.         I_B_f1.main();
                                               22.         .....
                                               23.     }
                                               24. } ;
```

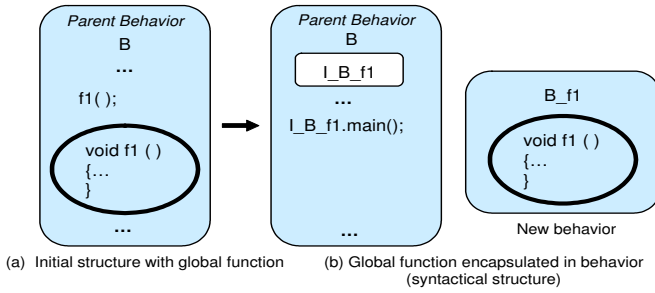(a) Original model (Model 1)   (b) Function encapsulated in behavior (Model 2)

Fig. 8. Encapsulating function into behavior

pointer in the interface, then using the pointer the block could access more than one variable at run-time. This ambiguity is overcome by using a flow-insensitive and context-insensitive pointer analysis [2] and replacing the pointer with the actual variable it points to [5]. In Figure 5 the *pointer* is replaced with the actual variable *s*. Note that the return value of the function (variable *c*) must also be considered as part of the interface (with write access type).

B. Encapsulating functions

The overview of re-coding involved in converting a function into a behavior is shown in Figure 7. The figure only shows the encapsulation aspect of structural hierarchy and not the connectivity aspect. The interface generated in the previous step is used to create the port list of the new behavior. Each port contains the direction information (*in*, *out*, *inout*) which is derived from the cumulative access information determined in the previous step. Figure 8(a) shows an example model with behavior *B* calling function *f1()*. This function is encapsulated into a new behavior *B_f1* as shown in Figure 8(b). The function call is re-scoped from its original scope into the new scope of the behavior (*B_f1*). The new behavior is created with the body containing a function call to the function (line 3 in Figure 8(b)).

After creating the new behavior, the behavior needs to be instantiated. The new behavior is instantiated (*I_B_f1*) in its parent behavior (*B*, line 13). The port map needed for the instantiation is generated by analyzing the function arguments and using the port list of the newly created behavior as reference. The port map for instance *I_B_f1* is *(a, b, i1, s, result)*. Also note that, the variables *a, b, i1, s,* which were originally in the local scope of function *B::main()*, are now re-scoped into *B*. This is necessary as these variables are needed for port mapping. After creating the instance, the original function call in parent behavior *B* is replaced with the call to the newly created instance (line 21).

C. Encapsulating statements

Similar to encapsulating functions, regular C statements can also be encapsulated. This transformation is necessary to encapsulate statements that exist between instances of behaviors so as to have a clean composition of behaviors at each hierarchical level. This transformation is similar to encapsulating functions and thus left out for brevity.

D. Establishing connectivity

Encapsulating functions and statements is just one aspect of structural hierarchy. After encapsulating the global functions,

the global variables in the global scope must be migrated from the global scope to a class scope where they are used. After doing so, since the variable is no longer global, explicit connection needs to be established by inserting ports in all the behaviors recursively across the entire hierarchy of behaviors. This transformation analyzes the access of the variables across the program and determines the lowest common parent scope. The variable is migrated into that scope and establishes the connection by inserting ports and parameters in all the behaviors and functions affected. This transformation is discussed in detail in [6].

IV. RECODING COMPLEXITIES

The automatic transformations must generate a model that is syntactically correct and semantically the same as the initial flat model. Though the program transformations described in the previous section seem straightforward and simple, there are complexities which, if not addressed, could result in incorrect code. Some of these complexities arise because of the difference in the semantics of the functions and behaviors. For example, the semantics of function parameters is different from the semantics of ports. When function parameters are replaced with ports, it is necessary to maintain the pass-by-value and pass-by-address semantics. This is ensured by adhering to strict recoding rules. For example, a function parameter passed as value can only be replaced with an *in* port irrespective of how the variable is accessed within the function body. Function parameters passed as address can be replaced with any of the ports as determined by CAT analysis.

Some of the complexities arise because of the programming style. Since expressions cannot appear in the portmap of an instance[3], expressions in function arguments such as *w+x* in line 3 of Figure 9(a) must be first evaluated into a temporary variable *wx* (line 6 in Figure 9(b)) and this temporary variable is used for portmapping (line 4). Similarly, when the return value from the function is ignored or read implicitly (line 3 Figure 9(a)), an explicit variable (*retval*) is created to hold the return value and all the implicit reads of the return value are replaced with the explicit read of this variable as shown by the

---

[3]Having an expression in the the portmap results in ambiguity regarding the location of the expression evaluation.

```
1. int func(int, int, int);          1. behavior B_func (in int, in int, in int, out int);
2. /*...*/                            2. /*...*/
3. if (func(w+x, y, z))               3. int wx, retval;
4. { /* do */                         4. B_func I_B_func (wx, y, z, retval); //Instance
5. }                                  5. /*...*/
                                      6. wx=w+x;
                                      7. I_B_func.main();
                                      8. if (retval)
                                      9. { /* do */
                                     10.}
  (a) Initial code with function func()   (b) Code after replacing func() with behavior
Fig. 9. Recoding complexities
```

modified *if* structure in line-3. When a variable in the local scope of the function is migrated into a class, it becomes a static variable and becomes available in the larger scope, thus becoming available to all the members in the scope. The transformation has to ensure renaming of the variable in case of name clashes.

The other complexities due to use of arrays and pointers in function calls are handled as described in Section B.

V. RESTRICTIONS

Though the transformations are automatic and handle most of the practical C codes, some programming constructs cannot be handled by our transformations. Encapsulating functions applies only to internal functions, as opposed to external/library functions. If the pointer analysis fails to determine the target variable, or if the pointer is determined to point to more than one variable, then the transformation is not performed. Further, encapsulating statements is difficult in presence of conditional *goto* statements which could transfer the control flow into the statement block under consideration.

VI. SOURCE RE-CODER

The structure of the SoC model depends on the underlying platform, application and also the architecture the designer has conceptualized. It is necessary to give control to the designer so that she/he can create a model that is most suitable to their needs. Further, in many C codes designer's inputs are critical in resolving many statically unanalyzable coding scenarios involving pointers, unstructured control flow (goto statements), recursive functions, and more. For example, in the context of introducing structural hierarchy, when there exists multiple function calls to the function being encapsulated, designer input is needed to decide the number of instances to be created. Meeting these requirements necessitates a designer-controlled environment, where the designer makes the design decisions and the tedious recoding happens through automation.

To aid the designer in coding and re-coding, we have integrated our transformations into a source *re-coder*. The source re-coder is a controlled, interactive approach to implement analysis and recoding tasks. In other words, it is an intelligent union of editor, compiler, and powerful transformation and analysis tools. The re-coder supports re-modeling of SLDL models at all levels of abstraction. It consists of 5 main components:

- Textual editor maintaining textual document object
- Abstract Syntax Tree (AST) of the design model to capture the structure of the program
- Preprocessor and Parser to convert the document object into AST

- Transformation and analysis tool set
- Code generator to apply changes

When the transformation to create structural hierarchy is invoked, the functional hierarchy of the input program is first presented to the designer. The designer invokes the automatic transformations on selected functions based on her/his knowledge of the application with a click of a button. The source code transformations are performed and presented to the designer *instantly* in the editor window. AST is designed to capture the complete structure of the program so that the code generator can generate the code in its near original form. The designer can also make changes to the code by typing and these changes are applied to the AST *on-the-fly*, keeping it updated all the time. This intelligent mix of application knowledge and the automation of the recoding makes our transformation very effective. Using source recoder, tedious and time-consuming manual programming is replaced by automatic programming.

VII. EXPERIMENTS AND RESULTS

We applied our source recoder on different real-life embedded C codes to create models with structural hierarchy. The transformation were implemented to create a well-structured model in SpecC [7] SLDL. First, we will demonstrate the use of source recoder on a MP3 decoder design example. The MP3 example had 30 functions and spanned around 3000 lines of code. Using the source recoder 43 behaviors were introduced to create the structured model. First the major functions were converted into behaviors, following which the C statements between them were encapsulated. Note that not all the functions were encapsulated into behaviors as some of the functions were too small and were called too often to be regarded as special computation blocks. An example code structure of part of the code segment and the corresponding structural hierarchy created using our source re-coder is shown in Figure 10.

The main advantage of creating structural hierarchy and making the model more analyzable is to enable automatic design exploration. To conduct automatic design exploration, we used the SCE tool-set [1]. The automatic refinement tool expects a model with clean structural hierarchy with all the computation blocks completely encapsulated. At every hierarchical level, the tool expects the behavior to contain either only C code (such behaviors known as leaf behaviors) or composed of behavior instances. Using one such structured SoC model of the MP3 decoder, we were able to evaluate 6 different HW/SW architectures using the SCE architecture refinement tool.

A. Productivity factor

Our source re-coder results in significant reduction in design time of the SoC model. To demonstrate this, we applied the source recoder on different industrial strength design examples. Four of these examples are listed in Table I. Each of these examples spanned few thousand lines of code. The table provides the number of functions in the input C code and number of behaviors that were introduced to create well-structured SoC model. The behaviors were created by encapsulating functions and statements. The functions for encapsulation were chosen based on our knowledge of the application. Small functions
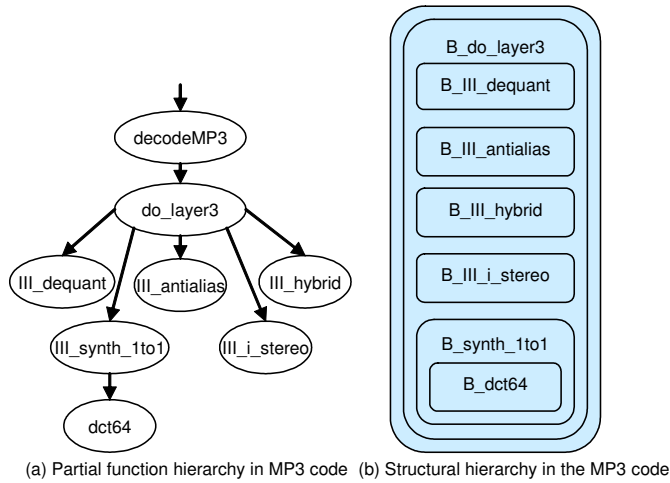
(a) Partial function hierarchy in MP3 code (b) Structural hierarchy in the MP3 code

Fig. 10. MP3 code strucutral hierarchy

TABLE I
PRODUCTIVITY GAIN FOR DIFFERENT EXAMPLES

Properties	JPEG	Float-point MP3	Fix-point 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 week	3 weeks	2 weeks	4 weeks
Productivity factor	120	205	120	192

which are called often, such as *getbits()*, were left as global functions. Using the automatic transformations in the source recoder, these models were created in a matter of minutes. In the past, these transformations were conducted manually on each of these examples by different designers. This manual recoding took weeks of development time as shown in Table I. Using our source recoder, the well-structured SoC models were created in the order of minutes instead of weeks, resulting in large productivity gains.

VIII. SUMMARY AND CONCLUSIONS

The lack of structural hierarchy and the presence of ambiguities makes the direct adoption of C code for system exploration and synthesis difficult. The design exploration and system synthesis tools require models with clean structural hierarchy, where the computation blocks are encapsulated and have a statically analyzable interface. The quality of this input SoC model directly determines the effectiveness of the system design tools. Creating this structural hierarchy and preparing the models for system synthesis is a critical and extremely time consuming task when performed manually.

In this paper, we proposed automatic source transformations to create models with structural hierarchy from the C code. The transformations use the existing partial structure available in the form of functional hierarchy to create behaviors with static interface. To control the structure of the model being generated, the transformations are made available to the designer in the form of an intelligent editor. The designer selectively chooses significant functions and statement blocks to be encapsulated and interactively invokes the transformation tools to realize the code transformations *on-the-fly*.

We showed that the transformations are effective on real-life

design examples. The original code with flat structure, which made the automatic architectural exploration tool ineffective, was transformed into a structured model which could facilitate exploration of multiple HW/SW partitionings. This automation of tedious recoding tasks and use of designer's knowledge makes our source recoder effective on real-life examples and results in large productivity gains.

REFERENCES

[1] S. Abdi, J. Peng, H. Yu, D. Shin, A. Gerstlauer, R. Dömer, 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.

[2] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994.

[3] P. Chandraiah and R. Dömer. Specification and design of an mp3 audio decoder. Technical Report CECS-TR-05-04, CECS, University of California, Irvine, 2005.

[4] P. Chandraiah and R. Dömer. Designer-controlled generation of parallel and flexible heterogeneous MPSoC specification. In *DAC*, 2007.

[5] P. Chandraiah and R. Dömer. Pointer re-coding for creating definitive mpsoc models. In *CODES*, 2007.

[6] P. Chandraiah, J. Peng, and R. Dömer. Creating explicit communication in SoC models using interactive re-coding. In *ASPDAC*, 2007.

[7] A. Gerstlauer, R. Dömer, J. Peng, and D. D. Gajski. *System Design: A Practical Guide with SpecC*. Kluwer Academic Publishers, 2001.

[8] A. Gerstlauer, S. Zhao, D. D. Gajski, and A. M. Horak. SpecC system-level design methodology applied to the design of a GSM vocoder. In *Proceedings of the Workshop of Synthesis and System Integration of Mixed Information Technologies*, Kyoto, Japan, April 2000.

[9] F. Ghenassia. *Transaction-Level Modeling with SystemC : TLM Concepts and Applications for Embedded Systems*. Springer-Verlag, 2006.

[10] International Telecommunication Union (ITU). *Specification and Description Language (SDL)*, November 1999. ITU-T Recommendation Z.100.

[11] A. Jerraya, H. Tenhunen, and W. Wolf. Guest editors' introduction: Multiprocessor systems-on-chips. *Computer*, 38(7):36–40, 2005.

[12] G. F. Marchioro, J.-M. Daveau, and A. A. Jerraya. Transformational partitioning for co-design of multiprocessor systems. In *ICCAD*, 1997.

[13] A. Pimentel, L.O.Hertzberger, P. Lieverse, and P. Wolf. Exploring embedded-systems architectures with artemis. *IEEE Transactions on Computers*, 34(1), November 2001.

[14] Sprint parallelizes real life applications for embedded systems. http://www.imec.be/design/sprint/.