Computer-Aided Recoding to Create Structured and Analyzable System Models

PRAMOD CHANDRAIAH and RAINER DÖMER
University of California, Irvine

In embedded system design, the quality of the input model has a direct bearing on the effectiveness of the system exploration and synthesis tools. Given a well-written system model, tools today are effective in generating working implementations. However, readily available C reference code is not conducive for immediate system synthesis as it lacks needed features for automatic analysis and synthesis. Among others, the lack of proper structure and the presence of intractable pointers in the reference code are factors that seriously hamper the effectiveness of system design tools. To overcome these deficiencies, we aim to automate the conversion of flat C code into a well-structured system model by applying automated source code transformations. We present a set of computer-aided *recoding* operations that enable the system designer to mitigate pointer problems and quickly create the necessary structural hierarchy so that the design model becomes easily analyzable and synthesizable. Utilizing the designer's knowledge, our interactive recoding transformations aid the designer in efficiently creating well-structured system models for rapid design space exploration and successful synthesis. Our estimated and measured experimental results show significant productivity gains through a substantial reduction of the model creation time.

Categories and Subject Descriptors: C.3.3 [**Special-Purpose and Application-based Systems**]: Real-Time and Embedded Systems

General Terms: Design, Algorithms

Additional Key Words and Phrases: system level design, embedded systems, synthesis, methodology, recoding

1. INTRODUCTION

The initial design model required by system-level design flows is an executable specification of the design, often called specification model [Gajski et al. 2000] or Transaction Level Model (TLM) [Ghenassia 2006]. Typically captured in a System Level Description Language (SLDL), this model directly determines the effectiveness of the applied synthesis tools and the quality of the end implementation. However, clean, well-written models suitable for automatic system design flows are not readily available. Instead, the Multi-Processor System-On-Chip (MPSoC) design process often starts from unstructured C code of the embedded applications. Easily obtained from open-source projects or standardizing committees, the application

Author's address: P. Chandraiah and R. Dömer, Center for Embedded Computer Systems, University of California at Irvine, Irvine CA 92697-2625, USA.

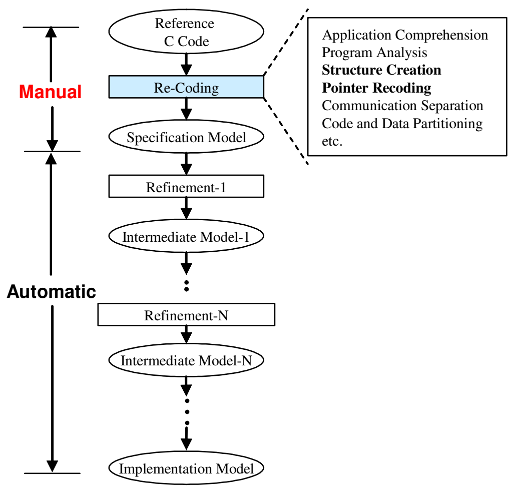


Fig. 1. Extent of automation in top-down system design flows.

code not only serves as reference for functional validation, but often is also used as starting model for deriving the end implementation.

However, there are significant obstacles in directly using such code for the system design process. First of all, these applications typically come from different domains, such as the signal-processing or general-purpose programming community. In particular, these sources are often designed and optimized as *implementations* to run fast on a regular single-CPU PC environment. As such, this code is usually not suited for embedded system design where a custom target architecture with multiple processing elements is to be used, and where a complex chain of tools (rather than a single compiler) generates the implementation. Second, the C language itself poses numerous challenges for system design. For a range of programming constructs, the freedom available makes the models ambiguous to system design tools. Commonly used constructs, such as pointers, dynamic memory allocation, recursion, and others, often result in ambiguities and negatively affect the analyzability, verifiability, and synthesizability of the model.

System design tools require well-defined and well-structured input models. For instance, architecture exploration, which partitions hardware and software blocks and maps these onto the architecture platform, usually requires an explicitly specified block structure in the model. Moreover, many system tools require models without pointers in order to enable static dependency analysis.

1.1 Scope of Work

In this work, we target the problem of preparing a well-written abstract system model for subsequent automatic design space exploration and system design, as illustrated in the top-down design flow shown in Figure 1. As is the case in many system design flows established today, we assume that the *Specification Model* serves as the reference for the hardware/software partitioning and the following refinement steps. However, this specification model typically must be written manually, even in the presence of available reference code.

Specifically, we address several issues that arise when the system designer creates the specification model in a SLDL, such as SystemC [Ghenassia 2006] or SpecC [Gajski et al. 2000], and starts with regular C reference code. Despite the similarity of C code and C-based SLDLs, manually *re-coding* the reference code into a suitable system model faces many obstacles, some of which are listed in Figure 1[1].

In this paper, we specifically address the creation of a suitable structural hierarchy (*structure creation*) and the elimination of unwanted pointers (*pointer recoding*). These and other coding tasks make manual system specification time-consuming, tedious, and error-prone.

The specification modeling phase is often a bottleneck in the overall system design flow. For example, we have used the top-down design flow in Figure 1 in a study on a MP3 decoder application [Chandraiah and Dömer 2005]. Since the refinement steps from the specification model down to the implementation model were mostly automated, we were able to implement our MP3 decoder in less than one week (down to RTL). In contrast, manually converting the MP3 reference code into a structured specification model took 12-14 weeks [Chandraiah and Dömer 2008b]. Thus, more than 90% of the overall design time was spent on writing and re-writing the initial specification model. Clearly, the specification phase was the main bottleneck in this design process.

1.2 Designer-controlled Approach

To tackle the specification bottleneck, our contribution in this work is an interactive, designer-controlled approach to create structured and analyzable system models where the system designer makes the decisions and the code is transformed automatically. Utilizing the designer's experience and application knowledge, we give her/him complete control over the specification process so that she/he can create a system model that is most suitable to the specific needs at hand.

Note that creating a system model involves hard problems that make the complete automation of this task infeasible. For example, to resolve statically unanalyzable code involving pointers, unstructured control flow (goto statements), recursive functions, or other issues, the human designer's intelligence is required. Also, the designer's application knowledge can help in reducing a problem to a solvable size. For example, the designer may want to resolve a few pointers that interfere in a specific portion of the code, while other pointers may remain unmodified in order to not negatively affect the readability of the original program.

For our designer-controlled approach, we have developed a *source recoder* which we describe in Section 5 in detail. Without disrupting the established system design flow, our source recoder introduces automation into the coding phase by integrating automatic source code transformations into an extended text editor. The recoder supports various transformations that let the designer quickly analyze the program, encapsulate communication, expose parallelism, restructure control flow, create structure, and eliminate pointers. As such, our interactive and extended editor proves effective in utilizing the designer's input to overcome the challenges posed by real-life reference code.

[1]Generally, the necessary source code modifications can be classified into structural and functional transformations, as well as analysis functions. An overview and brief discussion of such transformations is available in [Chandraiah and Dömer 2007a].

In essence, our interactive recoding process combines the intelligence of the designer with fast transformations by the tool and we rely on the designer to concur, augment, or overrule the automatic results to ensure the quality of the model.

Our work finds its application in preparing clean system models in C-based languages (i.e. plain ANSI-C as well as C-based SLDLs, such as SpecC and SystemC) at any abstraction level (i.e. at the design entry as well as at intermediate models in the design flow). Specifically, our recoder can be used as frontend editor in many system design flows (e.g. [Haubelt et al. 2008; Gerstlauer et al. 2008; Dömer et al. 2008]) and is instrumental as well for preparing suitable models for backend behavioral synthesis tools with no or limited support for pointer handling.

1.3 Related Work

The automated creation of structured system models from readily available reference applications is a problem that has not received much attention. This applies to all three system design approaches, top-down, bottom-up, and platform-based. For instance, the top-down design flows of SpecSyn [Gajski et al. 1994] and SpecC [Gajski et al. 2000] both require a well-defined specification model that needs to be written manually by the designer. Similarly, the component-based design flow [Cesario et al. 2002] starts with a virtual architecture model provided by the designer. As a commercial example, the platform-based VCC flow [Schirrmeister and Sangiovanni-Vincentelli 2001] expects a designer-written functional description of the application along with the platform architecture to generate performance models. The importance of a high-quality input model and the need for automation to create it is also emphasized in [Jerraya et al. 2005].

1.3.1 *Structure Creation.* Creating a structured specification model is addressed by several related works. For instance, [Marchioro et al. 1997] provide user-guided transformations for functional partitioning and structural reorganization to transform a system-level specification in SDL [International Telecommunication Union (ITU) 1999] into a hardware/software architecture in VHDL and C, respectively. In contrast, our work adds structure to flat C code and produces a hierarchical SpecC model.

The *Compaan* tool set [Pimentel et al. 2001] transforms a sequential application written in Matlab into a Kahn Process Network that acts as input model for architecture exploration of multiprocessor architectures for multimedia applications. *Sprint* [Cockx et al. 2007] transforms a sequential C program into a task-level pipelined program in SystemC with user-defined task boundaries. Unlike these, the key ingredient in our interactive approach is the control of the designer who is not restricted to a single type of model or predefined order of transformations. Instead, she/he has complete control to code/recode the model to arrive at the most suitable specification.

1.3.2 *Pointer Analysis.* Needed as a prerequisite to our pointer recoding, the topic of pointer analysis has been studied extensively over the last two decades. Traditionally, it is used by compilers to address data analysis problems like constant propagation and live variable minimization which are needed for program optimizations and error detection. Nevertheless, pointers pose serious challenges when programs meant for single-core single-memory architectures are used for creating system models of multi-core multi-memory platforms.

In general, precise pointer analysis is undecidable [Landi 1992; Ramalingam 1994; Hind 2001]. Existing algorithms trade-off between run-time efficiency and precision. The general problem of pointer analysis can be divided into two parts, *Points-To* and *Alias* analysis. *Points-to* analysis attempts to statically determine the memory locations a pointer can point to. On the other hand, *alias* analysis attempts to determine if two pointer expressions could point to the same memory location.

In the context of pointer recoding, we are primarily interested in *points-to* analysis. The research in this area is summarized very well in [Hind 2001]. Different pointer analysis algorithms differ in the precision of the analysis, efficiency of the algorithm, and scalability. Broadly, these algorithms can be classified based on two independent aspects, flow sensitivity and context sensitivity. Flow-insensitive algorithms [Steensgaard 1996; Zhang et al. 1996; Andersen 1994] do not consider the control flow of the program and hence are faster than flow-sensitive algorithms [Choi et al. 1993], that can potentially offer more precise results. Flow-insensitive analysis can again be broadly differentiated as unification-based [Steensgaard 1996] or inclusion-based [Andersen 1994], the former being faster but less precise.

The accuracy of such algorithms can be improved by adding context-sensitivity. A context-sensitive algorithm [Wilson and Lam 1995; Fahndrich et al. 2000] considers the effect of calling functions on the callee functions, and vice-versa. On the contrary, a context-insensitive algorithm is conservative and assumes that a callee affects all callers. Besides these two aspects, these algorithms differ depending on whether composite data is considered as one object or multiple individual objects. Further, high precision analysis of dynamically instantiated data structures requires shape analysis techniques [Ghiya and Hendren 1995].

1.3.3 *Pointer Recoding.* Although the problem of pointer analysis has been widely addressed by the compiler community, the problem of pointer *recoding* has not been addressed as such. SpC [Séméria and Micheli 1998] proposes a technique to synthesize pointers in hardware. Indirect reads and writes through pointers are replaced with direct accesses to variables and new control structures (if-else, switch-case). Pointer values are encoded and addresses are then removed. The control structures introduced are synthesized into multiplexers during behavioral synthesis. In contrast, our approach can be viewed as a pointer refactoring technique, rather than a synthesizing technique. Our pointer accesses are replaced with direct variable accesses without changing the control structure of the program. We keep the code close to its original form. Transformations at source level and keeping the code readable are, in our work, as important as the pointer recoding itself.

In the compiler community, pointer conversion has been studied with the goal of generating optimized code for processors. In particular, specialized techniques have been developed in order to recover closed-form representations from pointer-based array accesses. For example, [Franke and O'boyle 2003] present an approach that uses high-level source-to-source transformations for an array recovery technique that automatically converts certain pointer accesses in C code back to arrays. Built into an optimizing compiler and aimed at DSP applications, their array recovery shows significant execution speed improvements when combined with other code transformations, such as loop unrolling. As is the case in our approach, their technique benefits from the fact that explicit array accesses are generally easier to analyze and optimize than pointer operations. Their work also confirms that array recovery alone only yields mixed results, but can be very effective when combined with other transformations[2]. While [Franke and O'boyle 2003] aim at optimizing DSP applications for execution on single digital signal processors and their array recovery is built into an automatic compiler for optimized code generation, our approach targets general embedded applications on heterogeneous multi-processor architectures and is part of an interactive source code editor for efficient system modeling. Also, in contrast to the focus on array recovery, our approach also supports recoding of pointers to pointers.

Pointer recoding is critical for MPSoC modeling and design, where the sequential specification has to be split and mapped onto multiple processors and multiple memories. Eliminating pointers here simplifies the required dependency analysis between different blocks in the application and thus enables efficient partitioning and flexible mapping to the target architecture. Since in this process the decision making by the system designer is key in our interactive approach, we need pointers to be explicitly recoded and replaced with regular variables, so that the system designer can easily see the existing data dependencies in the application code. This is in contrast to optimizing compilers or synthesis tools, where data dependence testing is integrated into a fully automatic process. Therefore, advanced dependence analysis techniques, such as the chains-of-recurrences algebra [Engelen et al. 2004], which are very effective in compilers [Birch et al. 2006], are not suitable for our designer-controlled approach.

Often pointers also need to be removed from the system specification so that individual tools, which otherwise are not capable of handling pointers, can easily analyze, compile, synthesize, and refine the design model. Examples include academic High-Level Synthesis (HLS) tools, such as SPARK [Gupta et al. 2004], which do not allow pointers in their input model.

On the other hand, several Electronic System Level (ESL) and behavioral synthesis tools provide built-in support for handling pointers. Catapult C Synthesis [Mentor Graphics Corp. 2008] and C-to-Silicon Compiler [Cadence Design Systems, Inc. 2008], for example, support certain types of pointers (statically determinable ones) in their synthesis flow for hardware modules. While these tools target single (or few) rather small (up to hundreds of lines of code) hardware modules, our approach in contrast aims at the true system-level where large designs (i.e. several thousand lines of code) consisting of both hardware and software blocks are modeled. Moreover, our approach is interactive and fully designer-controlled, as opposed to these completely automatic synthesis tools. This allows the system designer to apply necessary pointer recoding transformations at specific instances, within limited regions of code, and at any time in any order (i.e. in between other code changes). In particular, by applying the pointer analysis only to a limited portion of the code, the system designer can even resolve and recode pointers for which static analysis fails when applied to the entire application code.

[2]This supports our argument to make our approach interactive and leave the decision to the user when and in which order to apply the recoding transformations.

1.4 Previous Work

In earlier work, we have proposed the recoding technique for code and data partitioning [Chandraiah and Dömer 2008c]. In this paper, we focus on creating comprehensible and analyzable models with proper structural hierarchy to facilitate design and exploration at the system level. We have introduced pointer recoding in [Chandraiah and Dömer 2007b] and structure creation in [Chandraiah and Dömer 2008b] in isolation and less detail. In this article, we not only address these topics comprehensively, we also present new results obtained from an experiment conducted with the help of a class of students. We further provide new architectural exploration results in Section 6.3 which show that both transformations together achieve more than each operation applied alone.

2. CREATING STRUCTURED AND ANALYZABLE MODELS

Before we discuss the proposed source code transformations in detail, we provide a motivating example that shows the creation of structural hierarchy and the necessary pointer recoding for creating a well-structured and analyzable system model.

As a common requirement, system-level design models clearly separate the computation and communication aspects of the application by using explicit constructs of the used SLDL. Computational blocks are encapsulated using modules (SystemC) or behaviors (SpecC), and communication between these blocks is captured using ports and channels. Unlike functions in a C program, computational blocks in SLDLs explicitly specify their interfaces by use of ports with defined direction (i.e. in, out, or inout). At the time of instantiation, the ports are statically mapped to variables. Thus, the interfaces of computational blocks and interconnecting channels are explicitly and statically defined. Synthesis and refinement tools therefore can easily determine the system connectivity and data flow.

```
short vec [10];
int a, b;
short *pvec;

int main (void)
{
  pvec = &vec[0];
  for(i=0; i<5; i++) {
    a = get32( );
    b = f1(a, pvec);
    pvec+=2;
  }
}




int f1(int a, short*p)
{ *p = a & 0xffff;
   p++;
  *p = a >>16;
  return *p;
}
```

(a) Original C-code:
Function with pointers

```
short vec [10];
int a, b;
short *pvec;
B_f1 bf1(a, pvec, b);
int main (void)
{
  pvec = &vec[0];
  for(i=0; i<5; i++) {
    a = get32( );
    bf1.main();
    pvec+=2;
  }
}

behavior  B_f1(in int a, inout
int*p, out int b)
{
  void main(void) {
    *p = a & 0xffff;
    p++;
    *p = a>>16;
    b = *p;
  }
};
```

(b) Model with pointers:
Behavior with pointer ports

```
short vec [10];
int a, b;
int ipvec;
B_f1 bf1(a, vec, ipvec, b);
int main (void)
{
  ipvec = 0;
  for(i=0; i<5; i++) {
    a = get32( );
    bf1.main();
    ipvec+=2;
  }
}

behavior  B_f1(in int a, inout short
vec[10], inout int ip, out int b)
{
  void main(void) {
    vec[ip] = a & 0xffff;
    ip++;
    vec[ip] = a>>16;
    b = vec[ip];
  }
};
```

(c) Recoded model:
Behavior with typed ports

Fig. 2. Example for encapsulation and pointer elimination.

To create a model with explicit structure from flat C code, designers typically convert given functions into behaviors[3] and derive the ports by analyzing the parameters and variables accessed in the functions. While this alone is already a tedious manual coding task, the presence of pointers in the C code aggravates the problem. Often pointer ports are created as a temporary solution. However, allowing pointer ports defeats the purpose of behaviors as it becomes possible to access more than a single variable through such ports.

Figure 2(a) shows an example where a function $f1$ with a return parameter p is converted to a behavior B_f1 with corresponding pointer port p (Figure 2(b)). Through this pointer the behavior accesses multiple variables which is confusing (for both designer and tools). Figure 2(c) shows the corrected design model which is obtained after replacing the pointer port p and array pointer *pvec* with an index *ip* and the actual array *vec*, respectively. Unlike Figure 2(a) and (b), the model in Figure 2(c) clearly reflects the actual interface of the behavior, easy to comprehend by the designer and easy to analyze by tools.

3. CREATING STRUCTURAL HIERARCHY

The C programming language is insufficient when one needs to explicity model structural hierarchy. C-based SLDLs, such as SystemC [Ghenassia 2006] and SpecC [Gajski et al. 2000], explicitly provide the necessary language extensions to specify the block-diagram structure and to separate computation from communication.

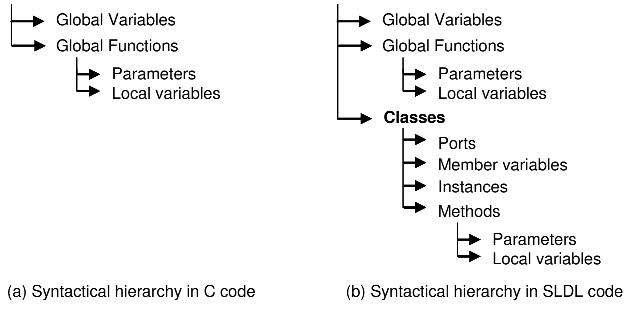


Fig. 3. Hierarchy of scopes in C and SLDL.

Syntactically, the SLDLs provide an extra level of scope, the *class* level, as depicted in Figure 3. C programs only have two levels, *global scope* and *local scope* inside functions. SLDLs, in contrast, provide an additional level of hierarchy, namely classes which represent modules, also known as behaviors, and channels. This *class scope* contains the ports, member variables, instances of other classes, and methods. Methods, just like functions, again have their own local scope consisting of local variables and parameters[4]. Explicit connectivity at the class level is available through ports.

Introducing structural hierarchy in SLDL models is a necessary step to enable architecture exploration and involves four tasks:

[3]For simplicity, we will use SpecC terminology (i.e. *behavior*) instead of SystemC (i.e. *module*) from here on. The approach and concepts, however, are equally applicable to both SLDLs.
[4]Minor levels of scope are also available as compound statement blocks in both C and SLDLs. These, however, are of very limited use and thus omitted in the figure for brevity.

(1) Encapsulating global functions into classes (by following the function call tree)
(2) Determining the variables affected by introduction of the new class scopes
(3) Migration of variables into the appropriate local, class, or global scope
(4) Establishing connections through channels, ports, and parameters

In the following, we will focus on the first task. For details on tasks (2) to (4), please refer to [Chandraiah et al. 2007].

3.1 Top-down Call-Tree Traversal

The function call tree in the C code can be used as a starting point to create an initial structural hierarchy in the design model. This way, major functions are encapsulated as separate behaviors. Using static analysis, we generate the function call hierarchy of the program. Figure 4(a) shows an example of a simple function call tree. Function *f1()* calls *f2()* which in turn calls *f3()* and *f4()*.

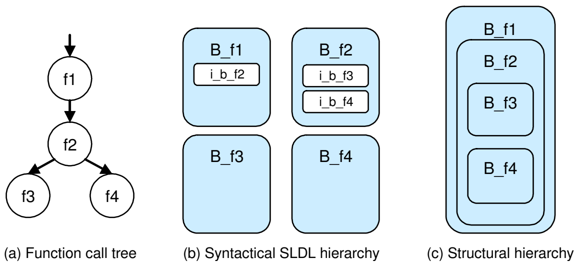


Fig. 4. Converting a function call tree to structural hierarchy.

We then traverse the call tree in hierarchical order and encapsulate the functions one by one in behavior shells with definite interface. Figure 4(b) shows the resulting syntactical structure after the functions are encapsulated. Here, `B_f2` is the declaration of a behavior containing `f2` and `i_b_f2` is an instance of that behavior. Figure 4(c) shows the final structural hierarchy of the resulting model.

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

(1) Determining the statically analyzable interface of the selected block of C code
(2) Encapsulating the block in a behavior/module
(3) Instantiating the new class and replacing the function call with a call to the new instance

We will now discuss each of these steps in more detail.

3.2 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), 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 fragment of C code and the corresponding interface of function *f*.

```
/* ... */
pa = &s;
result = f (a, b[i1], pa);
/*...*/
```

(a) Function call to *f*

Var.	Type	Access	Port Direction
a	int	R	in
b	int [10]	R	in
i1	int	R	in
s	char	RW	inout
r	int	W	out

(b) Interface of *f*

Fig. 5. Determining the statically analyzable interface of code blocks.

The complete interface includes the access type information, e.g. read, write, or read-write. This interface of a block is determined by analyzing the accesses of all variables in the block of statements. The overall access of a variable in a block is the accumulation of all local accesses in the individual expressions. Since we want the interface to be statically analyzable, for any vector accessed using a non-constant index variable, we make the safe assumption that the whole array is accessed[5]. If the access to the specific array elements cannot be determined statically, as for *b[i1]* in Figure 5, the complete interface will be defined to include the whole array *b* and the index *i1*. If there is a pointer in the interface, then the block can access more than one variable at run-time. This ambiguity is overcome using pointer recoding, which we will discuss later in Section 4. In Figure 5, the pointer *pa* is replaced with the actual variable *s*. Note that the return value of the function (variable *r*) must also be considered as part of the interface, but with write access type.

3.3 Encapsulating functions

The recoding involved in converting a function into a behavior is shown in Figure 6. 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*) and corresponds to the access type information determined in the previous step. Figure 6(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 6(b). The function call is re-scoped into the new scope of behavior *B_f1*. The new behavior is created with the body containing a call to the function (line 3 in Figure 6(b)).

After creation, the new behavior needs to be instantiated (*I_B_f1*) in its parent behavior (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)*. Note that the variables *a, b, i1*, and *s*, which were originally located in the local scope of function *B::main()*, are now re-scoped into *B*. This is necessary as these variables are needed for the 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 20).

[5]This conservative assumption could be replaced in the future by using advanced techniques for subarray access analysis.

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

(a) Original model (Model 1)

```
behavior B_f1( in int w, in  int x[10],
         in int i, inout int s, out int c) {
  void main()
  {  c= f1(w, x[i], &s);
  }
  int f1( int w, int x, int *p)
  {  *p  = w+x+*p;
     return *p;
  }
};
behavior B (in int p1, in int p2, out int result) {
  int a, b[10], i1, s;
  B_f1 I_B_f1(a, b, i1, s, result); // instance
  void main( )
  {
     int *pa;
     a = p1+p2;
     s = p1-p2;
     pa = &s;
     I_ B_f1.main();
  }
} ;
```

(b) Function encapsulated in behavior (Model 2)

Fig. 6. Encapsulating a function into a behavior.

3.4 Encapsulating statements

Similar to encapsulating functions, regular C statements can also be encapsulated. This transformation is necessary to encapsulate statement blocks that exist between instances of behaviors in order to obtain a clean composition of behaviors at each hierarchical level. This transformation involves the following steps:

—Creating a port list from the variable accesses
—Creating the behavior with the statement block as body
—Re-scoping variables that need to be port mapped
—Creating the instance of the behavior using the port map
—Replacing the statements with the call to the instance

3.5 Establishing connectivity

Encapsulating functions and statements is just one aspect of creating structural hierarchy. After encapsulating the global functions, variables in the global scope must be migrated to the class scope where they are used. After that, since those variables are no longer global, explicit connections need to be established by recursively inserting ports in all affected behaviors across the hierarchy. The details of this transformation are beyond the scope of this article and are discussed in detail in [Chandraiah et al. 2007].

3.6 Recoding complications

Our source transformations must generate a model that is syntactically correct and semantically equivalent to the input C program. Though the program transformations described in the previous section seem straightforward, there are complications which, if not addressed, would result in incorrect code. Additional problems arise because of the differences in the execution semantics of functions and behaviors.

For example, the semantics of function parameters are 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-reference 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

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

(a) 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. { /* do */
10.}
```

(b) Code after replacing *func()* with behavior

Fig. 7. Recoding complications when converting functions to behaviors.

within the function body. Function parameters passed as reference can be replaced with any of the port directions as determined by the analysis described above.

Other complications arise because of programming style. Since expressions cannot appear in the portmap of an instance[6], expressions in function arguments, such as *w+x* in line 3 of Figure 7(a), must be first evaluated and stored in a temporary variable *wx* (line 6 in Figure 7(b)) which is then used in the port mapping (line 4). Similarly, if the return value of a function is ignored or read implicitly (line 3 Figure 7(a)), an explicit result variable (*retval*) is created to hold the return value. All implicit accesses to the return value can then be replaced with an explicit read access to this variable, as shown by the modified *if* statement in line 8.

Finally, when a variable in the local scope of a function is migrated into a class, it becomes a static variable visible in a larger scope. In case of naming conflicts, our transformations have to automatically rename such variables and adjust all their accesses.

3.7 Limitations

Although our structure transformations handle most embedded C code, some programming constructs are not supported by our transformations. First, encapsulating functions applies only to function definitions (that have a body), as opposed to external library functions (functions declared *extern* without a body)[7]. Also, encapsulating recursive functions is not supported. Further, we do not support the encapsulation of statements in the presence of conditional *goto* statements which could transfer the control flow into the statement block under consideration.

Note that these limitations are not serious as such situations are rarely found in embedded applications.

4. POINTER RECODING

We will now address the elimination of unwanted pointers in the source code. Pointer elimination or replacement, which we will refer to as *pointer recoding* from here on, requires as a prerequisite the well-known task of *pointer analysis*[8] which we reviewed earlier in Section 1.3.2.

[6]An expression in a portmap results in ambiguity regarding the time of the expression evaluation.
[7]Note that the use of library functions, for example I/O functions, does not pose any problem. These functions only cannot be encapsulated as behaviors by themselves.
[8]Note that we are not proposing a new pointer analysis algorithm here. Pointer analysis is just a prerequisite to our pointer recoding.

In its concluding note [Hind 2001] correctly remarks that pointer analysis must be tailored to meet the accuracy, efficiency and scalability requirements of the client applications. Since our pointer recoding is interactive (for reasons stated in Section 1.2), run-time efficiency and scalability are an important concern[9]. Consequently, we selected flow-insensitive and context-insensitive points-to analysis for our recoder. Specifically, we chose an inclusion-based algorithm [Andersen 1994] over a unification-based algorithm [Steensgaard 1996], as the former provides more precision than the later. Andersen's algorithm offers reasonable precision and performance suitable for our pointer recoding. Unlike Andersen's approach, however, we have implemented our algorithm on an Abstract Syntax Tree (AST) representation of the design model (similar to the approach taken by [Buss et al. 2005]). The AST is generated from the description of the design model and captures the complete syntactical structure of the model. That is, it preserves all design information including blocks, functions, channels, ports, statements, expressions, and so on. It also includes code formatting details, such as line and column numbers of all objects, so that the code generator can reproduce the source description back in its original format.

```
1.  int a[50], ab[50][16];
2.  int v1, v2, x, y;
3.  int *p1,*p2, *p3, *p4,
    (*p5)[16], *p6;
4.  p1 = &x;
5.  *p1 = y + 1;
6.  if (condition)
7.      p2 = &v1;
8.  else
9.      p2 = &v2;
10. *p2 = 5;
11. p3 = &ab[40][10];
12. *p3 = 100;
13. p4 = a;
14. p4++;
15. *p4++ = 1;
16. p5 = &ab[5];
17. p6 = p4 + v1;
```

(a) Code with pointers

```
p1 → x
p2 → v1, v2
p3 → ab
p4 → a
p5 → ab
p6 → a
```

(b) Points-to list

```
1.  int a[50], ab[50][16];
2.  int v1, v2, x, y;
3.  int ip3, ip4,
    ip5, ip6;
4.  // nothing here
5.  x = y + 1;
6.  if (condition)
7.      p2 = &v1;
8.  else
9.      p2 = &v2;
10. *p2 = 5;
11. ip3 = 10;
12. ab[40][ip3] = 100;
13. ip4 = 0;
14. ip4++;
15. a[ip4++] = 1;
16. ip5 = 5;
17. ip6 = ip4 + v1;
```

(c) Pointers recoded

Fig. 8. Pointer recoding example.

An example points-to list generated by our analysis is shown in Figure 8. Our algorithm assumes that after incrementing a pointer to an array, the pointer still points to the same array. For instance, pointer *p4* points only to *a* despite being incremented[10]. Depending on the program, the points-to list of a pointer can contain one or more variables. In Figure 8, all pointers except *p2* bind to a single variable.

Our recoding is performed after all pointers are analyzed and bound to their variables. We perform recoding only on the pointers which bind to exactly one variable. If there is a possibility that a pointer could point to more than one

[9]A discussion of the responsiveness of our interactive source recoder and detailed execution times of recoding transformations are available in [Chandraiah and Dömer 2007a].

[10]Only in erroneous or non-portable programs will arithmetic on a pointer make it point to different variables.

variable (for example *p2*), then pointer recoding is not performed. Such pointers are brought to the designer's attention and the decision is left to the designer to resolve this issue. As described in Section 4.6, the designer can use his application knowledge and provide accurate binding information to facilitate recoding. Besides regular assignment expressions, our pointer analysis also takes into account pointer binding through port-mapping and function parameters.

4.1 Pointer Recoding: Background

Generally, pointers are problematic because they implement multiple concepts. A programmer can use a pointer as a *value*, *alias*, *address*, or an *offset*. It is a *value* when the absolute value of the pointer is used, it is an *alias* when it points to more than one variable in its life time, an *address* when it is simply dereferenced, and an *offset* when the pointer points into an array and is manipulated using pointer arithmetic. Pointer recoding can be performed automatically in the latter two cases, that is when the pointer is not aliased or used as an absolute value.

Recoding generally means to replace any indirect access to a variable through a pointer by a direct access to the variable. We distinguish scalar and array pointers, as follows:

—A pointer access to a scalar variable is replaced with the actual scalar. In Figure 8, this recoding applies to variable x accessed through pointer *p1*. Recoding mainly affects the dereferencing operation of *p1*, as shown in lines 4-5. The pointer initialization in line 4 is deleted as it is no longer necessary.

—For every pointer to an array, an integer is created which acts as index into the array. Then, the pointer access to vector variables is replaced with the array access operator '[ ]' using the actual vector variable and the newly created index variable. In Figure 8, this recoding applies to array variables *a* and *ab*. The newly created integers, *ip3, ip4, ip5* and *ip6*, are used as indices. Arithmetic operations on pointers are replaced with arithmetic on the index variables, as shown in lines 14, 15 and 17 in Figure 8(c). Pointer initialization is replaced with an initialization of the associated index variable with an offset expression (lines 11, 13, and 16).

While pointer recoding in Figure 8 may appear simple, it requires careful handling of all affected C operators and full support of complex expression trees.

4.2 Recoding Pointer Expressions

Given a pointer and its type, our algorithm recursively traverses the AST of an expression in a depth-first manner, searching for the specified pointer. Since each node in the AST has only local information, upon traversal, each node returns all information of interest to the node above. The parent node, using the results provided by each of its children, can then determine the proper course of action to take.

Upon traversing a node, our recursive recoding function returns a tuple of 4 elements {*e1*, *e2*, *e3*, *e4*}. *e1* contains the unmodified original expression. *e2* contains the expression of the index variable if the expression processed was a pointer, or an offset expression if the expression processed was a regular variable. If the expression processed is a pointer, *e3* contains the target symbol to which the pointer is bound to. *e4* is a Boolean flag indicating whether or not there was a positive pointer match.

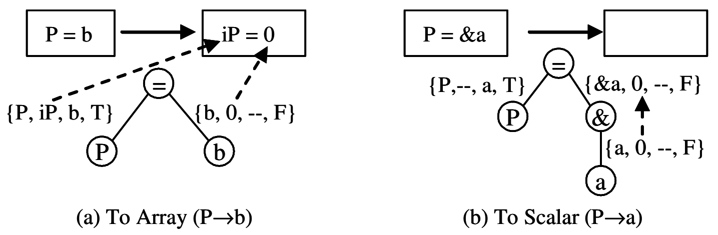


Fig. 9. Recoding pointer initializations.

To illustrate the algorithm in detail, we will now walk through the procedure to recode several pointer expressions. *Pointer initialization* to an array is shown in Figure 9(a). The figure shows how the algorithm operates on the AST when it is invoked to recode the pointer *P*, where *P* in this case points to vector *b*. Our recursive recoder starts from the assignment node ('=') and reaches the identifier *P*. The 4-element tuple returned by the node traversal is shown in the curly brackets. Since *P* is the pointer to be recoded, the original identifier (*P*), the index variable associated with the pointer (*iP*), the target variable the pointer binds to (*b*), and a Boolean asserting the pointer match (*true*) are returned as a tuple. Next, the other child node *b* is reached. Three elements, the original identifier expression *b*, an integer offset 0 (instead of an index variable, since *b* is not a pointer), and the Boolean flag *false* are returned. After returning to the *assignment* node, the pointer assignment is replaced with a new assignment expression formed using the index variable *iP* and the offset expression *0*.

In general, an assignment node (=) can receive 3 results (*e1, e2, e3*). The appropriate choice is selected depending on the *node type* and *e4*. If *e4* is *false*, *e1* is chosen. If *e4* is *true*, then the *node type* decides between *e2* and *e3*. If the *node type* is a pointer, *e2* is chosen, otherwise *e3*.

Recoding *pointer initialization* to a scalar variable is shown in Figure 9(b). Here, the pointer assignment expression is completely removed as the index assignment makes sense only for arrays. The necessary binding information (*P*→*a*) is maintained by the recoder.

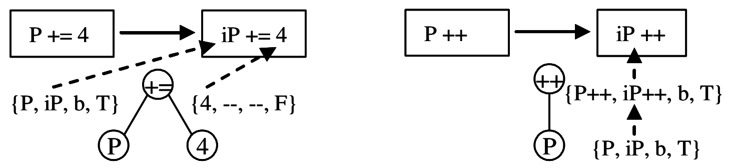


Fig. 10. Recoding pointer arithmetic expressions (P → b).

Pointer arithmetic, as shown in Figure 10, is replaced with arithmetic on pointer indices. Of course, this applies only to pointers to arrays. *Pointer assignment* in Figure 11 is similarily replaced with an assignment expression of the indices of the two pointers (*iP, iQ*).

While recoding *pointer dereferencing* expressions, three main scenarios need to be addressed (Figure 12), depending on the type of the target variable. If the

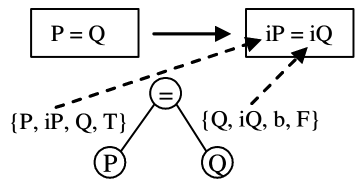


Fig. 11. Recoding pointer assignment (P → Q → b).

target is a scalar, the dereferencing node will return just the target scalar ({–, –, *a*, *T*}), as shown in Figure 12(a). If the target is an array, an array access expression (*b[ip]*), formed using the target array and the index variable of the pointer, is used as the replacement expression, as shown in Figure 12(b). Figure 12(c), a variant of Figure 12(b), shows recoding of the pointer expression that combines both dereferencing and pointer arithmetic.

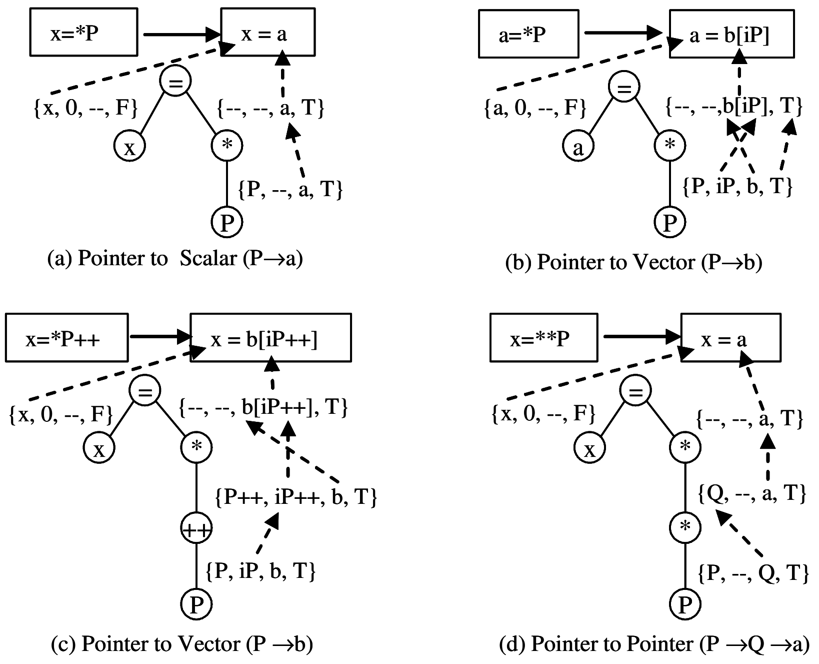


Fig. 12. Recoding pointer dereferencing expressions.

If the target variable is another pointer, all 4 elements, the target pointer, the index variable of the target pointer, the variable pointed to by the target pointer and the matching flag ({*Q,–,a,T*}), are returned. The next dereferencing node chooses a as the target, as shown in Figure 12(d). Note that whenever the information at the node is limited, as in case of expressions $++, +, -, /, *, \&, \|$ ..., no decision is made about choosing the correct result. Instead, all four results returned by individual child nodes are combined using the operator type of the current expression, as shown for *P++* in Figure 12(c), and passed to the parent node.

4.3 Recoding Pointers to Multi-dimensional Arrays

The offset expression generated while initializing the pointer to the beginning of an array is simply 0. However, initializing a one-dimensional pointer to a multi-dimensional array requires more attention. When a one-dimensional pointer is used to access a multi-dimensional array, properly replacing the pointers with the actual array variable requires separate index variables for each dimension. However, this would result in additional overhead because initialization and arithmetic on pointers will be translated into multiple initializations and arithmetic operations involving each index variable. To avoid this, we associate only one index variable with the pointer, based on the assumption that the pointer is used to point to only the elements across one dimension[11].

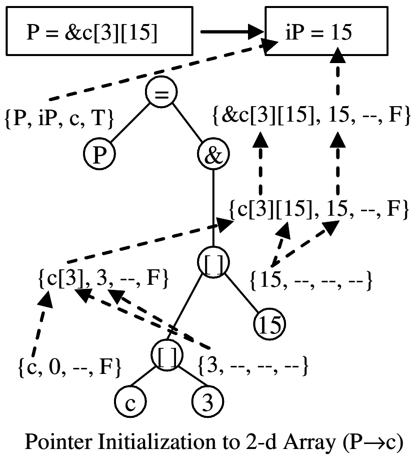


Fig. 13. Recoding pointers to multi-dimensional arrays.

For example, pointer P in Figure 13 is assumed to point only to 20 elements in row 3 of the array c. Thus, the index variable iP can only range from 0-19. More specifically, P is bound to sub-array *c[3]* and is used to replace any dereferencing expression of P. The offset derived is propagated upwards through the AST, as shown by the dotted arrows in Figure 13.

Figure 8 also shows multi-dimensional recoding for pointer *p3*.

4.4 Recoding Pointer Dependents

If it is determined that two pointers are dependent on each other, then recoding one of them requires recoding the other. For instance, if we are recoding pointer P and P depends on Q (as in expression $P = Q + 4$), then this requires recoding Q along with P. In our approach, the preparation stage identifies such dependent pointers and creates a list. The original and the dependent pointer are then iteratively recoded.

[11]Note that this is a safe assumption for proper ANSI-C code, even if pointer arithmetic is used that crosses from one dimension to the next.

4.5 Recoding Pointers in Function Arguments

Pointers that appear as function arguments also need recoding. A pointer argument is replaced with the target variable and, if applicable, the index variable of the pointer. A dereferencing pointer argument is replaced with just the variable it points to. Besides recoding the arguments, the corresponding function parameter must also be recoded to change the function signature. This recoding is scheduled and is recoded later along with the other dependent pointers (Section 4.4).

Note that by replacing pointer arguments with actual variables, a call-by-reference is changed to a call-by value if the pointer points to a scalar. Hence, immediately after this recoding, the program, though syntactically correct, is semantically not the same anymore. However, as soon as that function is encapsulated in a behavior (see Section 3.3), its parameters are converted to ports with direction information (in, out, inout) and proper semantics will be restored[12].

4.6 Interactive Designer Input for Robust Pointer Recoding

Making pointer recoding available to the designer as an interactive operation proves an efficient approach to extend the automatic analysis with the designer's intelligence and application knowledge. This way, our recoder can also resolve pointers that cannot be statically analyzed.

First, the designer can specify a limited context (a portion of code) where the pointer analysis and recoding are applied. A pointer, that is statically unanalyzable within the entire program context, often can be easily analyzed within a specific function, behavior, or segment of code. Thus, by allowing the designer to specify the recoding context it becomes possible to recode pointers that otherwise cannot be handled.

Second, if a pointer is determined to point to more than a single target variable, the application-aware designer can override the analysis and simply select the actual target variable to be used for recoding.

It is this type of interactivity that makes our proposed pointer recoding robust and effective for real-life embedded application code. Finally, involving the designer actively in the recoding process helps to ensure the quality of the design model[13].

4.7 Limitations

Though most practical pointer usages can be recoded by our approach, there are some limitations. We cannot recode pointers in the situations shown in Figure 14.

—Our recoder is meant only for pointers to static or stack variables, not for dynamically allocated memory.

—It is not possible to recode pointers whose values are being used as absolute value, for example, P in Figure 14 is read in line 5.

—We do not support pointer recoding to scalars if an arithmetic operation on the

[12]Another recoding solution to preserve the semantics would be to convert the target scalar into a vector of size one. However, the details of this alternative are outside the scope of this paper.

[13]At this point, our work enables automation in the design specification phase. Taking quality considerations and tradeoffs into account, for example different versions of transformations to satisfy different goals, is a next step that we leave for future work.

```
int A, B, *P, *Q;
char* R;
void* S;
P = (int*) malloc(10*sizeof(int));
if (P == 0)          // P is read by value, cannot be recoded
{ // code ... }
Q = &A;
*Q = 1;
Q++;                 // Q points beyond a scalar, cannot be recoded
R = (char*) (&B); // char pointer R points to an integer, cannot be recoded
*R = 0; R++; *R = 0;
S = (void*) (&B);  // void pointer S points to an integer, cannot be recoded
```

Fig. 14. Pointer recoding limitations.

pointer is performed. Pointer Q in Figure 14 shows this case. Note that such operations are not ANSI-C compliant.

—Operations involving different pointer types are not recoded. For example, pointer R, a character pointer, and S, a void pointer, are used to point to an integer.

Despite these restrictions, we find it possible to recode a large majority of pointers in practical sources. In our experience, most embedded applications obey proper ANSI-C coding guidelines for pointers and as such can be successfully recoded.

5. SOURCE RECODER

To aid the designer in coding and recoding, we have integrated our transformations into a *source recoder* [Chandraiah et al. 2007]. Our source recoder 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 recoder supports re-modeling of SLDL models at all levels of abstraction. It consists of 5 main components:

—Textual editor maintaining the textual document object

—Abstract Syntax Tree (AST) capturing the syntactical structure of the model

—Preprocessor and parser to convert the document object into AST

—Transformation and analysis tool set

—Code generator to apply changes

The designer can invoke the automatic source code transformations on selected objects simply by a click of a button. For example, to encapsulate a set of statements in a behavior, the designer highlights the statements in the editor window and invokes the transformation. Similarily, the designer can recode pointers with a click of a button, invoking the pointer recoder on individual pointers of her/his choice. All source code transformations are performed and presented to the designer instantly in the editor window.

Our AST data structure is designed specifically to capture the complete syntactical structure of the model so that the code generator can update the source code after transformations in near original form. Vice versa, the designer can also make changes to the code by typing and these changes are applied to the AST on-the-fly, keeping it synchronized at any time.

The interative mix of the designers' intelligence and application knowledge with the automated recoding transformations makes the design model creation and maintenance very efficient. Using the source recoder, tedious and time-consuming manual coding is replaced by automatic programming.

6. EXPERIMENTS AND RESULTS

To demonstrate the effectiveness of the recoding approach, we present the following experiments and results:

(1) We show the effectiveness of the pointer recoder.
(2) We describe a case study that creates a model of a MP3 audio decoder.
(3) To demonstrate the benefits of a structured and analyzable model, we present results of architecture exploration. In particular, we show how the creation of such models enables design tools in exploring additional architectures.
(4) We present a classroom case study to measure the productivity gains achieved by using our interactive recoder.

6.1 Effectiveness of pointer recoding

The main advantage of recoding pointers is to enhance program comprehension for the designer and to make the model conducive for tools with limited or no capability to handle pointers. Our interactive source recoder makes automatic pointer recoding feasible in many real-life embedded source codes. To show this, we have applied our pointer recoder to the embedded benchmarks listed in Table I [Chandraiah and Dömer 2007b]. Since operations, such as file I/O, typically become part of the testbench, we examined the above examples only in the context of the kernel functions listed.

Table I. Pointer recoding on different benchmarks.

Example	Applicable functions	Recoded pointers
adpcm [MiBench]	*adpcm_coder(), adpcm_decoder()*	6/6
FFT [MiBench]	*fft_float()*	0/4
sha [MiBench]	*sha_transform()*	1/1
blowfish [MiBench]	*BF_encrypt(), BF_cfb64_encrypt() BF_cbc_encrypt()*	10/10
susan [MiBench]	*susan_corners(), susan_principle() susan_edge()*	13/17
Float-MP3 decoder [MPG123]	*decodeMP3()*	14/16
Fix-MP3 decoder [MAD MP3 Decoder]	*III_decode(), synth_full()*	22/23
GSM	Across the program	17/17

For each example, Table I lists the number of pointers that our tool automatically recoded. The remaining pointers required user intervention. For example, in case of the *FFT* benchmark, all 4 pointers were being used as a *value* in a condition test and therefore could not be recoded. The majority of the other cases were pointers used for dynamic memory management (compare Section 4.7). Overall, however, our pointer recoder was effective and automatically recoded 83 percent of the pointers in the listed examples.

6.2 Creation of Structural hierarchy

We have applied our source recoder to different real-life embedded C codes to create models with proper structural hierarchy. The transformations were used to create a well-structured model in SpecC. Here, we will demonstrate the use of our source recoder on a MP3 decoder. The original MP3 example had 30 functions and spanned around 3000 lines of code. Using the source recoder, 43 behaviors were introduced to create a structured model. First the major functions were converted into behaviors, after which the C statements between them were encapsulated. Note that not all the functions were encapsulated into behaviors as some of them were too small and were called too often to be regarded as separate computation blocks.

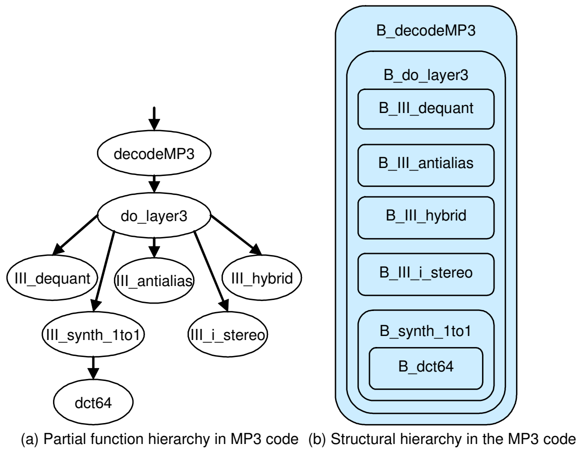


Fig. 15. MP3 structural hierarchy.

An example code structure and the corresponding structural hierarchy created are illustrated in Figure 15 [Chandraiah and Dömer 2008b].

6.3 Architectural Exploration

The main advantage of creating structural hierarchy and making the model statically analyzable is to enable automatic design space exploration. To conduct automatic exploration, we have used the SCE tool suite [Dömer et al. 2008]. The SCE refinement tools expect a model with a clean structural hierarchy where all computation blocks are properly encapsulated. At every hierarchy level, the tools expect the behaviors to contain either only C code (such behaviors are known as leaf behaviors) or be cleanly composed of behavior instances.

Using our source recoder, we have created two such clean system models by applying the two sets of recoding transformations discussed in this article. Specifically, we have created a model of a floating-point MP3 decoder, as shown in Figure 15, and a fix-point MP3 decoder. Using these two models, we were able to explore a total of 20 different HW/SW architectures with the SCE architecture refinement tool.

Even a sequential model with sufficient number of behaviors can result in several HW/SW architectures with varying performance. If the initial model is sequential (i.e. no explicit parallelism is specified), the performance benefits with different architectures come only through hardware acceleration[14].

[14]We have explored several parallel architectures in [Chandraiah and Dömer 2008c]. In this article, we present new sequential architectures that were made possible only because of the two recoding

Table II. Fixed-point MP3 decoder exploration.

Architecture	Components	Modules in HW	TLM	BFM	< 26.12 ms
Arch-1	ARM7TDMI (50 MHz),	none	48.62 ms	48.90 ms	—
Arch-2	ARM7TDMI (50 MHz), HW (100 MHz)	Stereo	47.08 ms	49.47 ms	—
Arch-3	ARM7TDMI (50 MHz), HW (100 MHz)	Alias	46.72 ms	49.99 ms	—
Arch-4	ARM7TDMI (50 MHz), HW (100 MHz)	IMDCT	41.07 ms	48.28 ms	—
Arch-5	ARM7TDMI (50 MHz), HW (100 MHz)	Stereo,Alias, IMDCT	37.64 ms	48.03 ms	—
Arch-6	ARM7TDMI (50 MHz), HW (100 MHz)	Synthesis Filter	15.93 ms	19.05 ms	OK
Arch-7	ARM7TDMI (50 MHz), HW (100 MHz)	Synthesis Filter, Stereo	14.39 ms	19.62 ms	OK
Arch-8	ARM7TDMI (50 MHz), HW (100 MHz)	Synthesis Filter, Alias	14.03 ms	19.81 ms	OK
Arch-9	ARM7TDMI (50 MHz), HW (100 MHz)	Synthesis Filter, IMDCT	8.39 ms	18.59 ms	OK
Arch-10	ARM7TDMI (50 MHz), HW (100 MHz)	Synthesis Filter, Stereo, Alias	12.49 ms	20.73 ms	OK

Table III. Floating-point MP3 decoder exploration.

Architecture	Components	Modules in HW	TLM	BFM	< 26.12 ms
Arch-1	Coldfire (66 MHz),	none	35.61 ms	35.61 ms	—
Arch-2	Coldfire (66 MHz), HW (66 MHz)	Stereo	34.84 ms	35.44 ms	—
Arch-3	Coldfire (66 MHz), HW (66 MHz)	Alias	34.87 ms	35.19 ms	—
Arch-4	Coldfire (66 MHz), HW (66 MHz)	IMDCT	31.65 ms	32.12 ms	—
Arch-5	Coldfire (66 MHz), HW (66 MHz)	Stereo,Alias, IMDCT	30.18 ms	31.50 ms	—
Arch-6	Coldfire (66 MHz), HW (66 MHz)	Synthesis Filter	22.73 ms	23.04 ms	OK
Arch-7	Coldfire (66 MHz), HW (66 MHz)	Synthesis Filter, Stereo	21.98 ms	22.85 ms	OK
Arch-8	Coldfire (66 MHz), HW (66 MHz)	Synthesis Filter, Alias	21.84 ms	22.43 ms	OK
Arch-9	Coldfire (66 MHz), HW (66 MHz)	Synthesis Filter, IMDCT	18.79 ms	19.25 ms	OK
Arch-10	Coldfire (66 MHz), HW (66 MHz)	Synthesis Filter, Stereo, Alias	21.09 ms	22.24 ms	OK

Table II and Table III list the different architectures that we explored for the fix-point and floating-point MP3 models, respectively. In both cases, *Arch-1* is the baseline architecture, namely a software-only implementation on a single processor. Specifically, we have used two embedded processor cores, an *ARM7TDMI* core for the fix-point, and a *Coldfire* processor for the floating-point version.

In order to increase the performance, we have added dedicated hardware accelerators in architectures *Arch-2* through *Arch-10*. Here, we explored the different design alternatives by mapping a different combination of modules (*IMDCT*, *Stereo*,

transformations combined in this paper

Aliasing, and *Synthesis filter*) to the hardware block.

We successfully refined each design alternative down to a Transaction Level Model (TLM) and a Bus-Functional Model (BFM) using SCE [Dömer et al. 2008]. Table II and Table III list the main components and their clock frequencies for each architecture. Except for the modules listed in the third column, which are mapped to hardware, the remaining parts of the decoder are mapped to the main processor.

Using the profiling capabilities in SCE, we have estimated the performance of each design alternative. The estimated time to decode one frame of MP3 data is given in the fourth and fifth column for the TLM and BFM, respectively[15]. Note that for a bitrate of 96000 bits/sec and a sampling frequency of 44.1 KHz, each audio frame must be decoded in less than 26.12 ms. For both the fixed-point and floating-point implementations, this timing constraint is met only by 5 out of the 10 possible architectures, as indicated in the last column of Tables II and III.

Note that the automatic exploration of these 10 successful architectures was made possible only due to the combination of pointer elimination and proper structure creation using our source recoder.

6.4 Productivity Gains

Our source recoder achieves a significant reduction in design time of the MPSoC model. To demonstrate this, we have applied our source recoder to different industrial-strength examples, three of which are listed in Table IV. Each example spans a few thousand lines of code. The table lists the number of functions in the input C code and the number of behaviors that we introduced to create a well-structured specification model. We created the behaviors by encapsulating functions and statement sequences that we chose based on our knowledge of the application.

Table IV. Transformations applied to different examples.

Properties	Floating-point MP3	Fix-point MP3	GSM
Lines of C code	3K	10K	10K
Lines of SpecC code	7K	13K	7K
C Functions	30	67	163
Behaviors created	43	54	70
Interfering pointers	16	23	17
Pointers recoded	14	22	17

Next, we recoded the pointer ports that otherwise would negatively affect the partitioning task. As shown in Table IV, 16 pointers interfered in creating a well-defined model of the floating-point MP3 code. Out of these 16, we eliminated 14 automatically by using our pointer recoder. Only 2 pointers could not be recoded automatically because (a) the absolute value of a pointer was read, and (b) a pointer pointed to more than one variable at run-time (see Section 4.7). We have recoded these two remaining pointers manually.

Each example was modeled by a different experienced designer. We collected the time taken to manually create behaviors from the reports of these designers, as listed in Table V. To obtain the time needed for manual pointer recoding, we first

[15]The BFM times are significantly higher as they contain communication delays which are ignored in the TLM.

Table V. Productivity gains for experienced designers.

Transformations	Time/Gain	Float-point MP3	Fix-point MP3	GSM
Behavior creation	Automatic recoding time	$\approx$ 35 mins	$\approx$ 40 mins	$\approx$ 50 mins
	Manual time	3 weeks	2 weeks	4 weeks
Pointer recoding	Automatic recoding time	$\approx$ 1 min	$\approx$ 1.5 min	$\approx$ 1.5 min
	Estimated Manual time	140 mins	220 mins	170 mins
Total gain	Productivity gain	203	120	189

recoded interfering pointers in different examples using Vim [Vim], an advanced text editor with block editing capability, and arrived at an average of about 10 minutes per pointer. The estimated manual time shown in Table V is obtained using this assumption[16]. In contrast, using the automatic transformations in the source recoder, these operations can be applied by the user in a matter of minutes[17]. This results in the high productivity gains listed in Table V.

6.5 Classroom Case Study

To obtain additional and more realistic productivity gains, we have conducted an experiment with a class of graduate students. The students were first given instructions to manually implement specific recoding tasks on a given MP3 decoder model. Next, we introduced the students to our source recoder and asked them to implement the same transformations using the automatic tool. For both tasks, the students were asked to measure the time taken for the various steps performed. Thus, in this classroom case study the same students have provided us with both manual and automatic times needed to implement the transformations.

6.5.1 *Setup.* A class of 15 students enrolled in a graduate course on SoC design [Dömer 2007] were instructed in general SoC description and modeling and, in particular, taught the SpecC SLDL. We then provided them with a MP3 audio decoder application modeled in SpecC. In a series of three assignments over a period of four weeks, we asked the students to implement specified recoding transformations for creating hierarchy and recoding pointers. In the first two assignments, the transformations were conducted manually, whereas in the third assignment we let the students apply the same transformations automatically using the source recoder.

In the first assignment, we asked the students to convert two function calls into behaviors. For the first behavior, the students were given detailed instructions to implement this transformation. For the second behavior, only brief instructions were provided. In order to measure the time needed to implement the transformation manually, the students were asked to report their time needed to successfully implement the transformations.

In the second assignment, we again provided the students with the source code of the MP3 decoder and asked them (1) to wrap two sets of C statements into behaviors, and (2) to perform pointer recoding on four specified pointers. For task (1), we provided detailed instructions for creating the first behavior and only brief instructions to encapsulate the statements for the second behavior. Similarly for task (2), we explained in detail the procedure to recode the first pointer and gave only brief instructions to recode the other three pointers. Again, we asked the stu-

[16]Table V covers only the pointers that our source recoder can automatically handle.

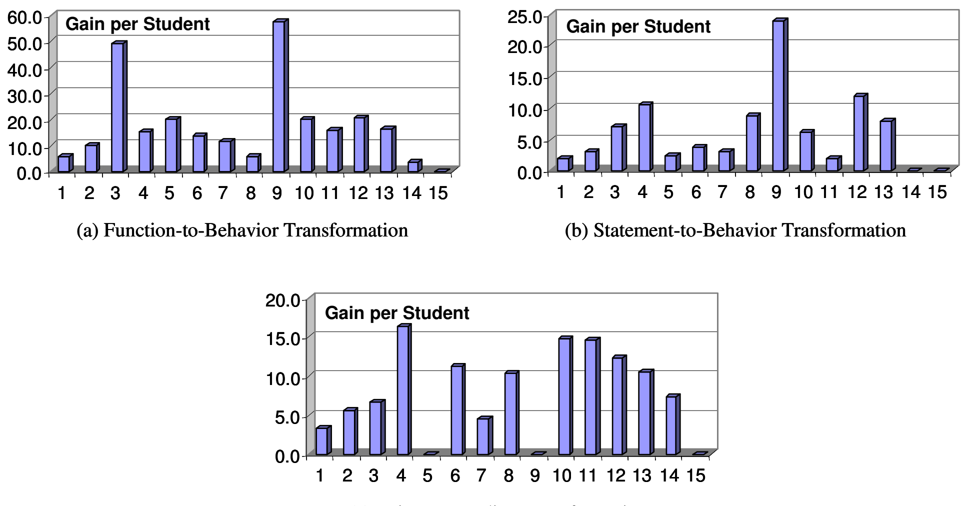
[17]The pure execution time of a pointer recoding transformation is below 1 second on a regular Linux PC (3 GHz Pentium-4) for all three applications.

dents to measure and report the times needed to implement these transformations successfully.

After completion of the two manual assignments, we introduced our source recoder to the students. In the third assignment, we then asked the students to implement the same transformations, that they manually implemented in the previous two assignments, using the automatic source recoder. The students again measured and reported the time taken to implement these transformations, however, this time using the tool.

For further details on this classroom case study, please refer to our extensive technical report [Chandraiah and Dömer 2008a] which documents the actual assignments and provides the obtained results in detail.

Note that in contrast to an ideal real-life design experiment, our classroom case study is admittedly limited to (a) a special group of test persons with mixed skills and experience (i.e. graduate students vs. actual system designers), (b) a short section of the overall system design process (we cover only part of the specification phase, not the entire design flow), and (c) a simple setup (for example, there is no control group). Despite these weaknesses, the measured results obtained here are complementary to estimated times (e.g. line 5 in Table V and results reported in [Chandraiah et al. 2007]) and therefore add significant value that supports the benefits of our computer-aided recoding approach.



(a) Function-to-Behavior Transformation

(b) Statement-to-Behavior Transformation

(c) Pointer-Recoding Transformation

Fig. 16. Productivity gains reported by different students in a classroom case study.

Table VI. Productivity factors determined in a classroom case study.

Recoding Transformations	Minimum	Average	Maximum
Function to Behavior	3.7	18.9	57.0
Statement to Behavior	1.4	8.1	60.0
Pointer Recoding	3.4	9.8	16.4

6.5.2 *Results.* Based on the times measured and reported by the students, we have analyzed and assessed the productivity factors that were achieved. Figure 16 plots the gains measured by different students for each of the three transformations, and the corresponding Table VI lists the average, minimum, and maximum productivity factors. Clearly, the gains achieved vary depending on the student[18], the type of transformation, and also due to our still imperfect source recoder[19]. Despite their variability, these factors show that our automatic transformations indeed result in significant productivity gains and are effective in reducing the time of system specification. Moreover, taking into account that the system specification time is a serious bottleneck in the design flow (recall the MP3 case study where over 90% of the time was spent in the specification phase [Chandraiah and Dömer 2005]), we can conclude that our approach significantly reduces the overall system design time.

Finally, we note that these measured productivity gains are lower than the estimated gains reported by the experienced designers in Table V. We attribute this to two reasons. First, this experiment accounted for the pure recoding time of these specific transformations. Second, the student designers were given line-by-line instructions for manual recoding which eliminates the otherwise necessary program comprehension and minimizes the typical coding errors and resulting debugging time. In the absence of such errors, the designers can direct all efforts and attention towards the actual modeling instead of textual recoding. This explains the higher gains reported by experienced designers.

7. CONCLUSIONS

In this work, we have introduced automatic source code transformations into the specification phase at the beginning of the MPSoC design flow. The needed input model is often created from available reference code of the embedded application at hand, which usually is not ready for immediate system synthesis. To create well-structured and analyzable input models, required code modifications include the addition of structure and the removal of pointers.

The contribution of this work is that we replace these lengthy manual code modifications by *computer-aided recoding* using automated source-to-source transformations. In [Chandraiah and Dömer 2008b], we have shown that manual code writing and re-writing is a bottleneck in the design process and can require over 90 percent of the overall design time. Using the automatic transformations presented in this work, this model specification time is effectively reduced.

In particular, we have presented two sets of automatic source code transformations in order to overcome (1) the lack of proper structure and (2) the presence of problematic pointers in existing application code. Aiming at the system level with large designs consisting of many hardware and software components, our *computer-aided recoding* approach enables the system designer to mitigate pointer problems and quickly create the needed block-based structure in the model.

(1) Creating a proper structural hierarchy and connectivity in the design model is critical in order to separate computation and communication and enable automatic design space exploration. We have presented several source code transformations that automatically encapsulate functions and statement sequences into instantiated

[18]Student 15 did not complete the assignments successfully.
[19]In a few cases, students reported bugs in the tool which we later fixed.

computation blocks with proper ports and connectivity.

(2) Our presented pointer recoding can in many cases eliminate unwanted pointers in the application code so that indirect variable accesses through pointers are automatically replaced by direct accesses to the actual target variables. This significantly improves the analyzability of the model by automatic tools and also aids the system designer in program comprehension. Being interactive, our pointer recoder can, with help of the intelligent designer, also resolve pointers that cannot be statically analyzed.

Our interactive source recoder augments complex automatic analysis and transformation tools by the designer's intelligence and application knowledge. In contrast to completely automatic synthesis tools, our fully designer-controlled approach allows the system designer to apply necessary pointer recoding and structure creation transformations at specific portions of the code, at any time, and in any order. The designer can invoke the automatic source code transformations on selected objects simply by a click of a button and the results are presented to the designer instantly in the editor window.

Our work finds its application in preparing suitable system models in C-based languages, i.e. regular ANSI-C as well as C-based SLDLs SpecC and SystemC. Design models can be recoded at any abstraction level, at the design entry as well as at intermediate stages in the design flow. Our source recoder can be used as a frontend editor for many system design flows and is instrumental also for preparing pointer-free models for backend synthesis tools with no or limited support for pointer handling.

We have shown that the proposed source code transformations are effective on real-life embedded application examples, some of which are of industrial size with several thousand lines of code. Our extensive experimental results show significant productivity gains, both in estimated improvements with the help of experienced designers, as well as in actually measured factors using a class of graduate students. In all cases, the results show significant productivity gains through a substantial reduction of the model creation time.

In future work, we plan to work on additional source code transformations. We also intend to apply our designer-controlled recoding approach to graphical representations of the design model. Finally, we plan to conduct a larger experiment with experienced designers and more transformations in order to overcome the limitations of our classroom case study.

8. ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their valuable suggestions to improve this article.

This work has been supported in part by funding from the National Science Foundation (NSF) under research grant NSF Award #0747523. The authors thank the NSF for the valuable support. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

REFERENCES

Andersen, L. O. 1994. Program analysis and specialization for the c programming language. Ph.D. thesis, DIKU, University of Copenhagen.

Birch, J., van Engelen, R., Gallivan, K., and Shou, Y. 2006. An empirical evaluation of chains of recurrences for array dependence testing. In *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques*. ACM, New York, NY, USA, 295–304.

Buss, M., Edwards, S. A., Yao, B., and Waddington, D. 2005. Pointer analysis for source-to-source transformations. In *SCAM*.

Cadence Design Systems, Inc. 2008. C-to-silicon compiler.

Cesario, W. O., Lyonnard, D., Nicolescu, G., Paviot, Y., Yoo, S., A.Jerraya, A., Gauthier, L., and Diaz-Nava, M. 2002. Multiprocessor soc platforms: A component-based design approach. *IEEE Design and Test of Computers*.

Chandraiah, P. and Dömer, R. 2005. Specification and design of an mp3 audio decoder. Tech. Rep. CECS-TR-05-04, Center for Embedded Computer Systems, University of California, Irvine. May.

Chandraiah, P. and Dömer, R. 2007a. An interactive model re-coder for efficient SoC specification. In *Proceedings of International Embedded Systems Symposium (IESS) Embedded System Design: Topics, Techniques and Trends (edited by A.Rettberg et.al)*. Springer.

Chandraiah, P. and Dömer, R. 2007b. Pointer re-coding for creating definitive MPSoC models. In *Proceedings of the International Symposium on Hardware-Software Codesign (CODES)*. Salzburg, Austria.

Chandraiah, P. and Dömer, R. 2008a. Assessment of productivity gains achieved through automated source re-coding. Tech. Rep. CECS-TR-08-02, Center for Embedded Computer Systems, University of California, Irvine. February.

Chandraiah, P. and Dömer, R. 2008b. Automatic re-coding of reference code into structured and analyzable soc models. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*. Seoul, Korea.

Chandraiah, P. and Dömer, R. 2008c. Code and data structure partitioning for parallel and flexible mpsoc specification using designer-controlled re-coding. In *Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*.

Chandraiah, P., Peng, J., and Dömer, R. 2007. Creating explicit communication in SoC models using interactive re-coding. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*. Yokohama, Japan.

Choi, J.-D., Burke, M., and Carini, P. 1993. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *POPL*.

Cockx, J., Denolf, K., Vanhoof, B., and Stahl, R. 2007. Sprint: a tool to generate concurrent transaction-level models from sequential code. *EURASIP J. Appl. Signal Process. 2007,* 1, 213–213.

Dömer, R. 2007. System-on-Chip description and modeling. *Lecture Notes for graduate course EECS222A*.

Dömer, R., Gerstlauer, A., Peng, J., Shin, D., Cai, L., Yu, H., Abdi, S., and Gajski, D. 2008. System-on-Chip Environment: A SpecC-based Framework for Heterogeneous MPSoC Design. *EURASIP Journal on Embedded Systems 2008,* 647953 (July), 13.

Engelen, R. A. v., Birch, J., and Gallivan, K. A. 2004. Array data dependence testing with the chains of recurrences algebra. In *IWIA '04: Proceedings of the Innovative Architecture for Future Generation High-Performance Processors and Systems*. IEEE Computer Society, Washington, DC, USA, 70–81.

Fahndrich, M., Rehof, J., and Das, M. 2000. Scalable context-sensitive flow analysis using instantiation constraints. In *PLDI*.

Franke, B. and O'boyle, M. 2003. Array recovery and high-level transformations for dsp applications. *ACM Trans. Embed. Comput. Syst. 2,* 2, 132–162.

GAJSKI, D. D., VAHID, F., NARAYAN, S., AND GONG, J. 1994. *Specification and Design of Embedded Systems.* Prentice Hall.

GAJSKI, D. D., ZHU, J., DÖMER, R., GERSTLAUER, A., AND ZHAO, S. 2000. *SpecC: Specification Language and Design Methodology.* Kluwer Academic Publishers.

GERSTLAUER, A., PENG, J., SHIN, D., GAJSKI, D., NAKAMURA, A., ARAKI, D., AND NISHIHARA, Y. 2008. Specify-explore-refine (SER): From specification to implementation. In *DAC '08: Proceedings of the 45th annual conference on Design automation.* 586–591.

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

GHIYA, R. AND HENDREN, L. J. 1995. Connection analysis: A practical interprocedural heap analysis for c. In *Languages and Compilers for Parallel Computing.*

GUPTA, S., GUPTA, R. K., DUTT, N. D., AND NICOLAU, A. 2004. Coordinated parallelizing compiler optimizations and high-level synthesis. *ACM Trans. Des. Autom. Electron. Syst. 9,* 4, 441–470.

HAUBELT, C., MEREDITH, M., SCHLICHTER, T., AND KEINERT, J. 2008. Systemcodesigner: Automatic design space exploration and rapid prototyping from behavioral models. In *DAC '08: Proceedings of the 45th annual conference on Design automation.* 580–585.

HIND, M. 2001. Pointer analysis: Haven't we solved this problem yet? In *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering.*

International Telecommunication Union (ITU) 1999. *Specification and Description Language (SDL).* International Telecommunication Union (ITU). ITU-T Recommendation Z.100.

JERRAYA, A., TENHUNEN, H., AND WOLF, W. 2005. Guest editors' introduction: Multiprocessor systems-on-chips. *Computer 38,* 7, 36–40.

LANDI, W. 1992. Undecidability of static analysis. *ACM Lett. Program. Lang. Syst. 1,* 4.

MAD MP3 DECODER. MAD (MPEG Audio Decoder) fix point mp3 algorithm implementation. http://sourceforge.net/projects/mad/.

MARCHIORO, G. F., DAVEAU, J.-M., AND JERRAYA, A. A. 1997. Transformational partitioning for co-design of multiprocessor systems. In *ICCAD.*

MENTOR GRAPHICS CORP. 2008. Catapult C synthesis.

MiBench. MiBench, A free, commercially representative embedded benchmark suite. http://www.eecs.umich.edu/mibench/.

MPG123. MPG123. http://www.mpg123.de/mpg123/mpg123-0.59r.tar.gz.

PIMENTEL, A., L.O.HERTZBERGER, LIEVERSE, P., AND WOLF, P. 2001. Exploring embedded-systems architectures with artemis. *IEEE Transactions on Computers 34,* 1 (November).

RAMALINGAM, G. 1994. The undecidability of aliasing. *ACM Trans. Program. Lang. Syst. 16,* 5.

SCHIRRMEISTER, F. AND SANGIOVANNI-VINCENTELLI, A. 2001. Virtual component co-design-applying function architecture co-design to automotive applications. *Vehicle Electronics Conference, 2001. IVEC 2001. Proceedings of the IEEE International.*

SÉMÉRIA, L. AND MICHELI, G. D. 1998. Spc: synthesis of pointers in c: application of pointer analysis to the behavioral synthesis from c. In *ICCAD '98: Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design.*

STEENSGAARD, B. 1996. Points-to analysis in almost linear time. In *POPL.*

Vim. Vim, advanced text editor. http://www.vim.org/index.php.

WILSON, R. P. AND LAM, M. S. 1995. Efficient context-sensitive pointer analysis for c programs. In *SIGPLAN PLDI.*

ZHANG, S. ET AL. 1996. Program decomposition for pointer aliasing: a step toward practical analyses. In *SIGSOFT.*

Received June 8, 2009; revised October 28, 2009; accepted TBD