Center for Embedded Computer Systems
University of California, Irvine

RIDE: Recoding Integrated Development Environment

Rainer Dömer

Technical Report CECS-13-02
April 29, 2013

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-2625, USA
(949) 824-8919

doemer@cecs.uci.edu
http://www.cecs.uci.edu

RIDE: Recoding Integrated Development Environment

Rainer Dömer

Technical Report CECS-13-02
April 29, 2013

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-2625, USA
(949) 824-8919

doemer@cecs.uci.edu
http://www.cecs.uci.edu

Abstract

One solution to address the steadily growing complexity of embedded computer systems is the modeling at higher levels of abstraction using System-Level Description Languages (SLDL) such as SpecC or SystemC. However, writing such executable system models is error-prone and extremely time-consuming. Little has been done to support system designers in the tedious manual coding and re-coding tasks necessary in embedded system specification. With the Recoding Integrated Development Environment (RIDE), we aim to automate the process of describing embedded systems by use of advanced computer-aided design (CAD) techniques. Proper analysis and automation of the tasks involved in system modeling will allow to shorten the design time significantly. Also, quality improvements in the end design are expected if automation relieves the system designer from complex code analysis and tedious coding tasks, allowing uninterrupted focus on the critical tasks of system modeling and design space exploration. RIDE follows an advanced modeling technique called designer-controlled re-coding *which largely automates the process of the creation of the system model. Using a combination of automatic and interactive transformations simultaneously on the source code and a graphical model representation, the proposed re-coding technique efficiently derives an executable parallel system model directly from available sequential reference code. Using a designer-controlled approach, the system designer utilizes his application knowledge and design experience, whereas the re-coder, an intelligent union of editor, compiler, and powerful transformation and analysis tools, realizes the complex analysis and model transformation tasks.*

Contents

List of Figures

List of Tables

RIDE: Recoding Integrated Development Environment

Rainer Dömer
Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-2625, USA
doemer@cecs.uci.edu
http://www.cecs.uci.edu

Abstract

One solution to address the steadily growing complexity of embedded computer systems is the modeling at higher levels of abstraction using System-Level Description Languages (SLDL) such as SpecC or SystemC. However, writing such executable system models is error-prone and extremely time-consuming. Little has been done to support system designers in the tedious manual coding and re-coding tasks necessary in embedded system specification. With the Recoding Integrated Development Environment (RIDE), we aim to automate the process of describing embedded systems by use of advanced computer-aided design (CAD) techniques. Proper analysis and automation of the tasks involved in system modeling will allow to shorten the design time significantly. Also, quality improvements in the end design are expected if automation relieves the system designer from complex code analysis and tedious coding tasks, allowing uninterrupted focus on the critical tasks of system modeling and design space exploration. RIDE follows an advanced modeling technique called designer-controlled re-coding *which largely automates the process of the creation of the system model. Using a combination of automatic and interactive transformations simultaneously on the source code and a graphical model representation, the proposed re-coding technique efficiently derives an executable parallel system model directly from available sequential reference code. Using a designer-controlled approach, the system designer utilizes his application knowledge and design experience, whereas the re-coder, an intelligent union of editor, compiler, and powerful transformation and analysis tools, realizes the complex analysis and model transformation tasks.*

1 Introduction

As we enter the information era, embedded computing systems have a profound impact on our everyday life and our entire society. With applications ranging from smart home appliances to video-enabled mobile phones, from real-time automotive applications to communication satellites, and from portable multi-media components to reliable medical devices, we interact and depend on embedded systems on a daily basis.

Over recent years, embedded systems have gained a tremendous amount of functionality and processing power and, at the same time, can now be integrated into a single Multi-Processor System-on-Chip (MPSoC). The design of such systems, however, faces great challenges due to the huge complexity of these systems. The system complexity grows rapidly with the increasing number of components that have to cooperate properly and in parallel. In addition, expectations grow while constraints are tightened. Last but not least, customer demand constantly requires a shorter time-to-market and thus puts high pressure on designers to reduce the design time for embedded systems.

1.1 Motivation

The design roadmap [30] of the international Semiconductor Industry Association ITRS predicts a significant productivity gap and anticipates tremendous challenges for the semiconductor industry in the near future. The 2004 update of the roadmap identifies system-level design as a major challenge to advance the design process. *System complexity* is listed as the top-most challenge in system-level design. As first promising solution to tackle the design complexity, the ITRS lists *higher-level abstraction and specification*. In other words, the most relevant driver to address the system complexity challenge is to raise the abstraction level. This is also emphasized in [28].

1.2 Significance of Modeling

In system level design, the importance of abstract modeling cannot be over-emphasized. Proper abstraction and modeling is the key to efficient and accurate estimation and successful implementation. However, in contrast to the great significance of abstraction and modeling, most research has focused on tasks *after* the design specification phase, such as simulation and synthesis. Little has been done to actually address the modeling problem. Difficulty is certainly one major obstacle. Also, the quality of a model is not straightforward to measure and compare.

A model is an abstraction of reality. More specifically, an embedded system model is an abstract representation of an actual or intended system. Only a well-designed model will accurately represent and define the properties of the end product, while allowing efficient examination and effective implementation.

Moreover, embedded system models must be executable. Execution of the model allows to simulate the behavior of the intended system and to measure properties beyond the immediate ones present in the model description. For example, simulation enables the prediction of properties such as performance and throughput.

The choice of the proper abstraction level is critical. Ideally, multiple well-defined abstraction levels are needed to enable gradual system refinement and synthesis, adding more detail to the design model with every step. In other words, a perfect model retains only the essential properties needed for the job at hand, and abstracts away all unneeded features. Then, as the design process continues, incrementally more features are added to the model.

1.2.1 Architecture analogy

The architectural blueprints of a house can serve as a good analogy that shows the importance of efficient modeling.

An architect, charged with designing a new building, typically develops a set of models of her/his intended design in order to examine, document and exhibit the intended building features, the general floorplan, room sizes, etc. For example, to exhibit the aesthetic qualities of the design, the architect often builds an abstract paper model of the building that shows the three-dimensional design in small scale. For the actual building phase, however, a different model is needed. Here, the architect draws two-dimensional schematic blueprints for each floor which accurately show the location and dimensions of the walls, doors and windows, as well as water and gas pipes, etc.

In this analogy, the purpose of the abstract models is clear, as is the inclusion or omission of design features. Different models serve different purposes and therefore exhibit different properties. Moreover, the quality of the architectural model determines the quality of the resulting building.

Another critical aspect in this analogy is the use of computer-aided design (CAD) tools when designing the blueprints and models of a building. CAD tools with advanced 3D graphics are used to create, modify, and analyze building models, which in turn leads to efficient blueprint development and high-quality buildings.

In contrast, most embedded designers today still specify their system models using basic text editors, the equivalent of pencil and paper for building architects!

1.3 Related Work

Program transformations have been used for different areas, including to improve aesthetics, to parallelize applications, and to perform high level synthesis. Examples include SUIF [18], Spark [17], Artemis [26], and Sprint [31]. Interactive program transformations of varying complexity are integrated into editors and programming environments, including Eclipse [23], MSVC [25], D-Editor [19], Parascope [21] and SUIF Explorer [22]. However, these are insufficient in creating MPSoC models. Code refactoring [13] is a software engineering technique used in type-safe object-oriented programming languages. In contrast, our transformations target MPSoC design and are interactive and designer-controlled.

2 Problem Definition

In essence, the modeling and parallelization of embedded system models suffers from two main problems. First, modeling and parellelizing a system in a System-Level Description Language (SLDL) is extremely time-consuming and considerably delays the development process. This creates a significant negative impact on design productivity. Second, most system-level design models today lack the quality needed for effective system synthesis, thus produce poor end designs. In other words, a specification model must be of highest quality, because its characteristics directly determine the quality of the end product.

2.1 Coding Bottleneck

In the past, the system-level design community has focused on solving various problems of system synthesis. Researchers have been working towards design automation at various abstraction levels with the goal to automate steps in the design process and reduce the design time. Motivated by the need to meet the time to market and aggressive design goals like low power, high performance, and low cost, researchers have proposed various new methodologies based on system design automation. These technological advances have significantly reduced the development time of embedded systems. However, *design time is still a bottleneck* in the development of embedded systems.

One critical aspect neglected by the EDA community so far is the design specification phase where the intended design is captured and modeled for the model-driven design flow. Each design methodology expects a very specific design model for the tools to be successful. These models need to be either hand-written from scratch or manually modified from a reference model. While much of the research has focused on synthesis and refinement tools, little has been done to support the designer in forming these models.

Fig. 1 shows a generic top-down system design methodology [14]. Following this system design flow, we have studied several industrial-size design examples, including a MP3 audio decoder [2] and a GSM voice codec [15]. For the MP3 design example, Fig. 1 shows the time spent for creating the design model in comparison to the refinement time using automated synthesis tools. Manually re-coding the reference implementation into a proper specification model took 12-14 weeks, whereas the entire implementation was completed in less than a single week of time. This and other examples confirm that *about 90% of the system design time is spent on coding and re-coding of the model*, even in the

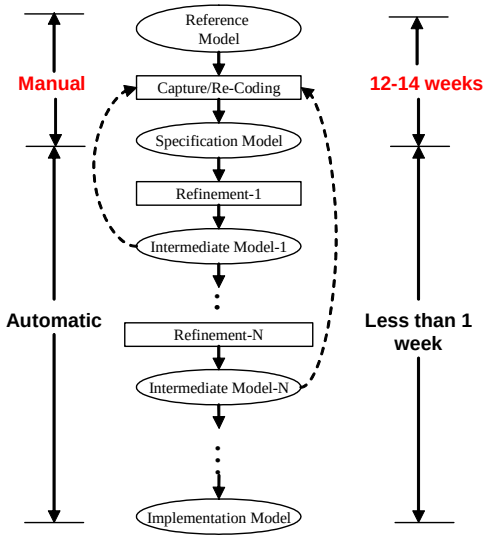


Figure 1: Design time and extent of automation in a refinement-based design flow [3].

presence of available reference code.

Moreover, we need to emphasize that model capturing is not a one time task. When a change in the design model is required for a refinement step down in the design flow, it is most often necessary to modify or *re-code* the input model. Such interruptions in the design flow cause costly delays.

2.2 Model Requirements

The quality of the system specification model has a direct bearing on the effectiveness of the system exploration and synthesis tools. The *quality of the model* is determined by its analyzability, flexibility, and potential for concurrency.

Analyzability of the model is essential for automatic synthesis and verification. If the model cannot be statically analyzed, tools are often unusable or ineffective.

Flexibility in the model is necessary for design space exploration. In system design, flexibility increases with the separation of computation and communication in the model. Moreover, flexibility depends on proper model granularity which determines on the amount of possible design partitions, i.e. mappings to processors, hardware units, memories, and system busses.

Explicitly specified *concurrency* in the model is required in order to efficiently utilize the parallel target MPSOC architecture. Whether the target platform consists of a multi- or many-core architecture, the available parallelism can only be exploited if it is apparent (explicitly expressed) in the model.

In summary, the time-consuming coding and re-coding of system models is a serious bottle-neck in the system design flow, and, at the same time, the quality of the model is most critical for successful implementation by use of automated tools.

3 Recoding Methodology

In the following, we describe the envisioned *Recoding Methodology* and show some initial results.

3.1 Computer-Aided Re-coding

Since the quality of the design model has tremendous impact on the cost and quality of the resulting system implementation, creating and optimizing the model of the intended system is a critical task in the design process.

Our methodology is called *designer-controlled re-coding* and utilizes a *computer-aided* approach. This technique efficiently reduces the time of modeling through interactive automation [7, 8, 3, 4, 5].

3.1.1 Re-coding approach

Creating a well-written embedded system model involves separation of communication and computation, introducing proper granularity, and exposing concurrency. In addition, it is also necessary to realize the model in a SLDL, such as SystemC [16] or SpecC [14], as only these can capture systems containing both hardware and software components at different levels of abstraction.

In [9] and [2], we have manually converted a C reference implementation of a MP3 audio decoder into a properly structured MPSoC specification in SpecC SLDL. The series of transformations started with the creation of a testbench that separates the design under test (DUT) from the stimulus and monitor blocks used for simulation. Next, we introduced structural hierarchy in the design by encapsulating major functions and enclosing statements. Then, we localized global variables by restricting their scope, in order to expose any hidden communication in the design. Finally, we exposed potential concurrency in the design by forming parallel and pipelined computational blocks. Only after all these changes, we finally arrived at a suitable specification model ready for use in our system synthesis flow.

Omitting any time needed for learning, we measured that the entire process of converting the given reference C code into a synthesizable SLDL description required 12-14 weeks for one full-time designer [3]. Clearly, design automation is needed to speed up this tedious coding and re-coding process.

However, to enable automation, we need to distinguish between *decision-making tasks*, that require human intelligence and experience, and *automatic tasks* that can be programmed and automatically executed. For example, higher-level tasks, such as determining task-level parallism in the design, or choosing the ”right” set of functions for encapsulation, need to be performed by the experienced designer based on his application knowledge. On the other hand, smaller tasks, such as adding a port to a block, routing a signal through the design hierarchy, and re-scoping a variable, can be automated by use of proper data structures and CAD algorithms.

More specifically, tedious textual re-coding operations *can* be performed automatically, *if* the decisions to apply these operations are made by the designer. This way, the designer is relieved from mundane text editing tasks and can focus on the actual modeling decisions.

Another way to look at the recoding task, is the observation that today, regardless of their complexity, all modification operations are performed manually by the designer using a regular text editor. However, there is a large discrepancy between the task the designer wants to perform (i.e. encapsulating a function and its arguments into a behavior block with ports), and the commands the text editor offers (i.e. adding and deleting characters, lines, or text blocks). Thus, the idea of designer-controlled recoding is to put powerful analysis operations and source code transformations at the disposal of the designer, such that she/he can apply modeling decisions to the design automatically, and instantly by a ”click of a button”.

To summarize our approach, *computer-aided recoding*, unlike other program transformation approaches, keeps the designer in the loop and provides complete control to create and modify the design model. By making the modeling process interactive, we rely on the designer to concur, augment or overrule the automatic analysis results of the tools, and use the combined intelligence of the system designer and the advanced analysis and transformation tools for the coding, re-coding, and modeling tasks.

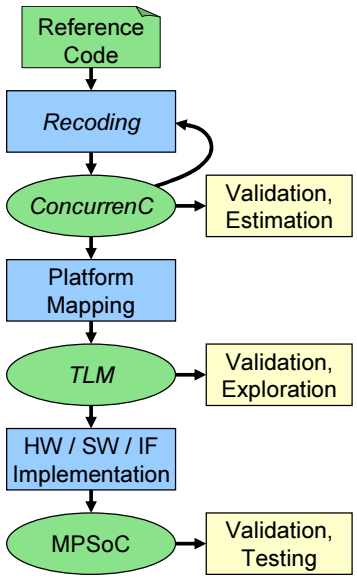


Figure 2: Envisioned design flow.

3.1.2 System design flow

In order to properly evaluate the proposed recoding techniques, we will integrate them into our existing system design flow. Fig. 2 shows the overall envisioned design flow and the position of the recoding tool.

Our design flow starts with available C-based reference code that is readily available (often times for free) for most applications (e.g. JPEG, MPEG, MP3, H.264, etc.). This allows to jump-start the development with real code that is accurate and usually well-tested. We then use computer-aided recoding to generate an initial platform-agnostic design model for early system validation and estimation. Next, after platform capture and mapping [27], we generate a transaction-level model (TLM) for design space exploration. For the final MPSoC implementation, we can then rely on our existing hardware, software, and interface synthesis tools [12].

3.2 RIDE: Recoding Integrated Development Environment

To aid the designer in system specification and modeling, we propose an Integrated Development Environment (IDE) for model re-coding. The *Recoding IDE* (RIDE) supports the designer-controlled interactive approach to automated coding and recoding as outlined above.

RIDE tightly integrates interactive graphical and textual editors with the system-level tool chain for simulation, estimation, refinement, and synthesis. In other words, RIDE is an intelligent union of editor, compiler and powerful transformation and analysis tools.

RIDE supports re-coding of SLDL models at various levels of refinement and at different stages in the design flow. It can be used to re-code intermediate system design models, as well as the initial C reference implementation in order to produce a parallel and optimized system model.

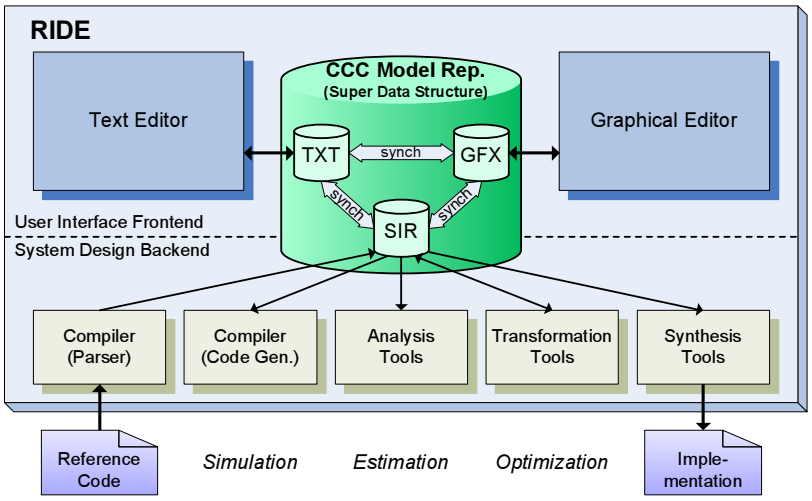


Figure 3: Recoding Integrated Development Environment.

The conceptual structure of the Recoding IDE is shown in Fig. 3. On the highest level, RIDE consists of a user interface frontend, a complex data structure for model representation, and a backend of advanced system design tools.

3.2.1 RIDE frontend

The RIDE frontend offers two main editors to the system designer, a graphical and a textual one. The textual editor maintains the SLDL source text of the design model and allows the usual navigation, modification, and editing tasks of modern text editors. Advanced features include syntax highlighting, auto-completion, semantic search, ctags, text folding, bookmarks, and undo/redo. Syntax and semantic support is provided for C-based languages, i.e. C and C++ programming languages, and SystemC [16] and SpecC [11] SLDLs.

The graphical editor presents a hierarchical diagram of the design model that can be used for visualization, navigation, and modification operations. Supported visualization operations include zoom/unzoon, selection of hierarchy depth, display of connectivity (ports, busses, channels), and highlighting of objects. On the other hand, graphical editing is supported by rename, add, move, delete, and cut/copy/paste operations for blocks (modules/behaviors), channels, interfaces, variables, ports, and connections.

Both editors are linked and constantly synchronized. Any change applied to the design in one editor instantly is reflected in the other editor as well. In other words, both editors maintain the same design model, and just display the model in a different perspective (textual view, graphical view). Note that the synchronization of the editors is instantaneous, on a key stroke or click of a button, and is accomplished by use of an advanced super data structure (see Section 3.2.2 below).

The RIDE frontend also controls the backend design tools using a convenient graphical user interface (GUI) with the usual tool bars, menu structures, and dialog windows. Moreover, the RIDE backend tools can be called directly from the design objects shown in the editors. Thus, when the designer points to and highlights any object, such as a channel icon in the graphical editor or a channel name in the textual interface, a context menu pops up with applicable and available operations. For example, when the designer points to a channel instance, the list of possible operations contains renaming, copying, deleting, changing the scope, and finding dependents and connected ports.

To facilitate an efficient implementation, we can use the Eclipse [23] IDE as base framework. Some early

experiments confirm that most (if not all) features listed above can be implemented as regular plug-in modules.

3.2.2 RIDE super data structure

At the core of the Recoding IDE, a complex data structure maintains a comprehensive, coherent, and consistent (CCC) model representation. The CCC model representation is a super data structure that combines three dedicated data structures, a textual model representation (TXT), a graphical model representation (GFX), and a syntax-independent representation (SIR).

The textual model representation (TXT), also known as the document object, represents the design model as program text specified in a system-level description language (SLDL). Ideally, RIDE would support both SystemC [16] and SpecC [11] SLDLs. In other words, the TXT representation is implemented by use of a regular text data structure used in a text editor, essentially consisting of lines of text, and augmented by support for ctags, syntax-highlighting, and other advanced features (which the Eclipse framework supports).

The graphical model representation (GFX) is a complex object-oriented data structure that describes the graphical hierarchy chart of the design in form of its coordinates, sizes, colors, and so on. Here, one can use the existing facilities in the Eclipse framework as basis, possibly augmented with special graphics by use of the commercial Qt GUI libary [1] (by Trolltech [20]).

The syntax-independent representation (SIR) [32, 10] is the central data structure for compilation, analysis, and transformation tools. It contains an abstract syntax tree (AST) of the design model that corresponds to the textual representation (TXT). In addition, the SIR contains full-fledged type and symbol tables, necessary to support compiler tasks such as parsing, semantic checking, static analysis, optimization, and code generation. Note that this data structure also maintains source code comments and position information so that a code generator can re-generate the source code for use in the TXT representation. Most notably, the SIR data structure is the basis for analysis tools toward early system estimation, for transformation tools toward re-coding, modeling and optimization, and for synthesis tools toward the final system implementation.

The three basis data structures, TXT, GFX and SIR, are combined into a RIDE super data structure that keeps the design model accurately reflected and updated in each representation. Sychronization functions (*synch* in Fig. 3) are used for this purpose. Any change to one of the three data structures is immediately synchronized with the others, such that after each modification or transformation, all data structures consistently reflect the resulting model.

We would like to emphasize that this instant synchronization of the different data structures becomes a key part of RIDE. While it is ambitious to maintain a comprehensive, coherent, and consistent super data structure, it is certainly feasible, even if synchronization has to occur frequently, i.e. after every key stroke in the text editor. As we see below in Section 4, an early prototype implementation of two synchronized data structures (TXT and SIR) showed sufficient responsiveness, even though an ad-hoc file interface was used for the synchronization operations.

3.2.3 RIDE backend

The RIDE backend is envisioned as a powerful set of analysis and transformation tools that the designer can invoke directly from the two editors. The results of these operations are directly reflected in the editors as well. In other words, the analysis and transformation tools build the core of the re-coding environment.

To provide an overview about envisioned analysis and transformation tasks, we can categorize the re-coding operations into three classes.

Analysis functions provide static analysis, such as dependency information, on the objects in the model without introducing any changes to it. As such, analysis can, for example, provide information to the designer about potential for parallel execution of blocks and/or functions. Conceptually, analysis function include

- revealing dependencies,
- check for potential concurrency, and
- general analysis for program comprehension.

Example operations in this category include determining the usage of variables, introducing concurrency, and generating dependency graphs.

Structural transformations change the structure of the design model by introducing and/or removing computational blocks, channels, and functions. In this category, we further distinguish

- granularity transformations,
- composition transformations,
- re-organizing transformations, and
- connectivity transformations.

Introducing new blocks (modules/behaviors), composing hierarchical blocks, splitting and/or merging blocks (to adjust the design granularity) are some examples of structural transformations.

Functional transformations modify computational blocks, functions, and variables. This category can be further subdivided into

- transformations to contain communication,
- transformation to break dependencies, and
- pruning transformations.

Localizing global variables, breaking composite data types into smaller data types, and trimming the width of wider data types into optimized bit vectors, are some examples of transformations in this category.

4 Preliminary Results

To demonstrate the feasibility of the proposed RIDE approach, and to estimate the productivity gains that can be expected, we have implemented a simple textual recoder [3] and obtained promising preliminary results that we will review in the following sections.

4.1 Source Recoder

In an early implementation, we have implemented a source re-coder [3] based on an existing QT [20] and Scintilla [29] based textual editor. In this extended text editor, the designer can call a small set of analysis and transformation operations by clicking on added buttons in the Graphical User Interface (GUI) provided by the editor.

Fig. 4 shows the software architecture of our preliminary source re-coder, consisting of 5 components, a text editor, a data structure (abstract syntax tree, AST), a preprocessor and parser, a code generator, and a set of initial transformation tools.

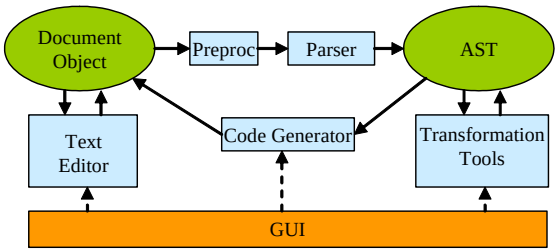


Figure 4: Early Source Re-Coder Structure.

Compared to the envisioned Recoding IDE in Fig. 3, this early implementation has no support for graphical editing (no GFX data structure), and lacks many other aspects. Most notably, the synchronization functions between the two data structures are implemented in a simple ad-hoc fashion. After changes in the text editor, the AST data structure is recreated by writing all source code into a file, and calling the C preprocessor and the SLDL parser on that file. In other words, the AST is built from scratch every time the text changes. Vice versa, the entire text file is replaced with a new file generated by a code generator when the AST changes due to a transformation operation.

Clearly, this ad-hoc implementation needs major improvement. Nevertheless, it allows us to gauge the feasibility of RIDE and even estimate the benefits and expected productivity gains.

4.2 Communication Recoding

Communication exploration is critical in MPSoC design. Given a design model with explicit communication, design space exploration can be conducted automatically. However, the communication in the design needs to be clearly exposed in the model by using and connecting standard communication channels. This, unfortunately, requires significant time in re-writing the model.

In [8], we have introduced our re-coding approach that is based on decision making

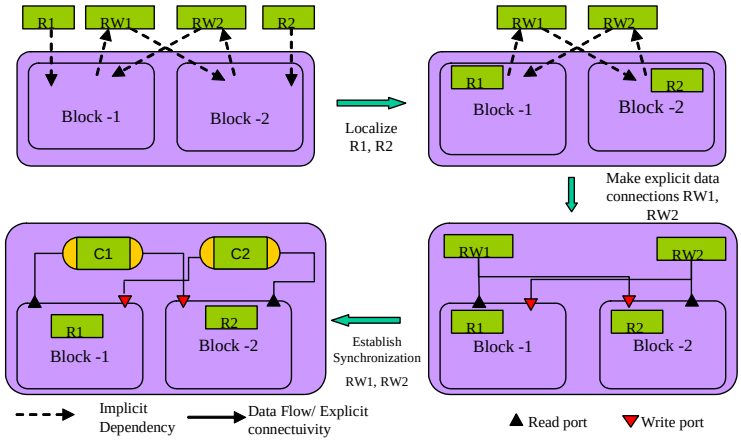


Figure 5: Recoding to create explicit communication.

by the designer (”designer-in-the-loop”) and automation of the model analysis and transformation tasks. In particular, we have developed 3 code transformations to expose communication, namely (1) localizing global variables, (2) establishing explicit connectivity, and (3) introducing synchronization. Fig. 5 shows these transformations in an overview.

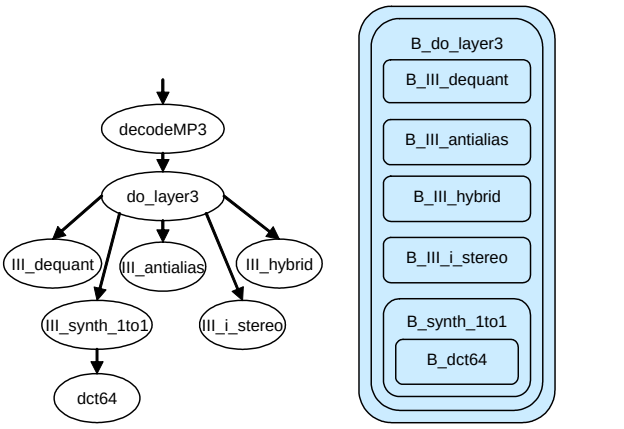
We have implemented these communication recoding transformations in our preliminary source recoder, and have estimated the resulting productivity gain in comparison with manual editing. Table 1 shows the results for three real-life examples. For each example, we have used our re-coder to expose the communication in the models, which involves the transformations (1) through (3). Table 1 shows the times measured using the re-coder compared to estimated manual times. It also shows the resulting productivity gains, more than 100x for each example.

Properties	JPEG	MP3	GSM
Global Variables localized	8	70	83
New Ports added	2	146	163
New Channels added	1	6	2
Re-coding time (secs)	27	246	260
Estimated Manual time (mins)	53	497	585
Productivity gain	**117x**	**121x**	**135x**

Table 1: Productivity gains for communication recoding [8].

4.3 Hierarchy Recoding

Structural hierarchy is a critical property needed in system design models. With a well-structured model, design exploration can evaluate various partitions by grouping and re-grouping different blocks and mapping them onto different components in the target architecture. The lack of structural hierarchy and the presence of ambiguities prohibits the direct use of flat C code. Design exploration and system synthesis tools require models with clean structural hierarchy, where all the computation blocks are properly encapsulated and have a statically analyzable interface. The quality of this input SoC model directly determines the effectiveness of the system design tools. Thus, creating a suitable structural hierarchy of behavioral blocks is critical. Moreover, it is also very time consuming when performed manually.



(a) Partial function hierarchy in MP3 code (b) Structural hierarchy in the MP3 code

Figure 6: Recoding to create structural hierarchy.

In [6], we have developed a set of automatic source code transformations that allows to create models with structural hierarchy from C reference code. The transformations use the given functional hierarchy to create a behavior tree with static interfaces. For the example of an MP3 decoder application, the creation of structural hierarchy is illustrated in Fig. 6.

Following our recoding methodology, the designer selectively chooses significant functions and statement blocks to be encapsulated and interactively invokes the automated code transformations.

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	$\approx$ 30 mins	$\approx$ 35 mins	$\approx$ 40 mins	$\approx$ 50 mins
Manual time	1.5 week	3 weeks	2 weeks	4 weeks
Productivity factor	**120x**	**205x**	**120x**	**192x**

Table 2: Productivity gains for hierarchy recoding [6].

Using our source recoder, we have applied these transformations to a set of industrial design examples, as listed in Table 2. Each of these examples spanned a few thousand lines of code. The table provides the number of functions

in the input C code and the number of behaviors that were introduced to create a well-structured MPSoC model. The behaviors were created by encapsulating functions and statements, chosen based on our application knowledge. Using the automatic transformations in the source recoder, the models were created in a matter of minutes.

Earlier, very similar transformations were conducted manually on these examples by different designers. This manual recoding took weeks of development time, as shown in Table 2. Using our source recoder, the well-structured SoC models were created in the order of minutes instead of weeks, resulting in large productivity gains.

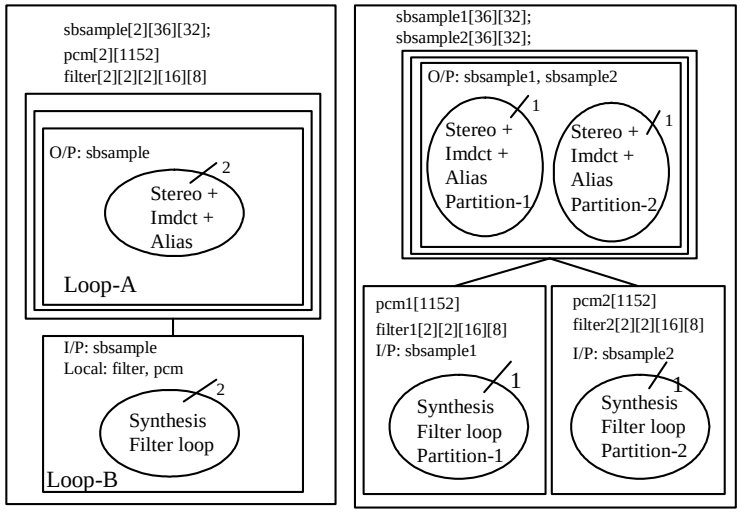
4.4 Parallel and Flexible Recoding

Concurrency and flexibility are two critical features of a MPSoC model. Concurrency in the model is necessary to exploit the parallel resources available on the multi- or many-core platform. Flexibility is necessary for freedom in design space exploration.

However, two main factors limit today's compilers in generating a parallel and flexible MPSoC specification automatically from a sequential monolithic application: first, the heterogeneous nature of MPSoCs with customized processors and non-regular memory hierarchy, and second, the complexity of the unstructured input application. Completely automatic compilers, though successful in extracting instruction level parallelism on shared-memory architectures, cannot expose task level parallelism which requires application-specific knowledge.

In contrast, we propose to create parallel and flexible models based on iterative designer-controlled transformations, instead of a monolithic completely automatic compilation. The discrete transformation steps are combined by the designer to create the desired specification model.

In [7, 4], we have proposed a designer-controlled approach to create a parallel and flexible MPSoC model. In particular, we have developed a set of six code and data partitioning transformations that can split loops and composite variables to expose concurrency and create flexibility. Our transformations include loop splitting, vector and composite data structure partitioning, variable localization, and synchronization of shared data. These transformations implement re-coding tasks that are intuitive even to a programmer with limited compiler knowledge.



(a) Sequential MP3 code structure (b) Parallel MP3 code structure

Figure 7: Recoding to create a flexible and parallel MP3 decoder.

Fig. 7 illustrates the data parallelism exposed in a MP3 decoder using our source re-coder. With the model created in Fig. 7, we could explore two distributed parallel design alternatives and two shared-memory based parallel architectures.

Table 3 show the results for the different fix-point MP3 implementations. For each design, the tables list the main components and their clock frequencies, as well as the performance achieved in decoding one frame of MP3 data. Note that a frame must be decoded in less than 26.12 ms. This timing constraint is only met by 5 out of the 10 possible architectures, as shown in the last row of the Table 3. Note that the required performance was met only due to the explicit parallelism exposed by our transformations. This clearly shows that the parallelism exposed through our trans-

formations is indeed effective.

4.5 Pointer Recoding

Models	Arch-1	Arch-2	Arch-3	Arch-4	Arch-5
	ARM7TDMI (50MHz)	ARM7TDMI (50MHz) 1 HW (100MHz)	ARM7TDMI (50MHz) 1 HW (100MHz) Shared Mem.	ARM7TDMI (50MHz) 2 HW (100MHz)	ARM7TDMI (50MHz) 2 HW (100MHz) Shared Mem.
TLM	48.62 ms	32.12 ms	32.12 ms	17.27 ms	17.27 ms
BFM	48.90 ms	33.83 ms	36.08 ms	20.33 ms	21.23 ms
$<$ 26.12 ms	—	—	—	OK	OK

Table 3: Flexible and parallel architectures of MP3 decoder. [7].

Due to the wide availability of C reference applications, the design of today's embedded systems often starts from C reference code typically obtained from open-source projects and standardizing committees. These C models are reused to create a system model in the desired SLDL. Though this code reuse speeds up the design process, it poses numerous challenges. The presence of pointers in the input C code is one such issue.

The ambiguity introduced by the use of pointers in the C code presents serious problems to system design tools. Most of today's synthesis and verification tools are not designed to handle pointers, as their primary goal is to address other tasks. To overcome this limitation, designers invest significant time and effort to create definitive unambiguous system models by recoding pointers.

```
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) p2 = &v1;
7.  else p2 = &v2;
8.  *p2 = 5;
9.  p3 = &ab[40][10];
10. *p3 = 100;
11. p4 = a;
12. p4++;
13. *p4++ = 1;
14. p5 = &ab[5];
15.  p6 = p4+v1;
```

(a) Code with pointers

```
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) p2 = &v1;
7.  else p2 = &v2;
8.  *p2 = 5;
9.  ip3 =10;
10. ab[40][ip3] = 100;
11. ip4 = 0;
12. ip4++;
13. a[ip4++] = 1;
14. ip5 = 5;
15. ip6 = ip4+v1;
```

(b) Code with p1, p3, p4, p5, p6 substituted

Figure 8: Pointer recoding example.

In [5], we have developed pointer recoding that can replace pointers in C code with actual variables. Our transformation recodes most pointer expressions and can recode even pointers that are used across functions and behaviors. Fig. 8 shows a typical pointer recoding example.

In general, resolving pointer ambiguity is a hard problem and a complete solution is not available. Hence, to be effective on real-life examples, pointer recoding needs to be interactive. By following our designer-controlled recoding methodology, the designer can selectively recode problematic pointers in the desired scope, and realize the code transformation *on-the-fly*. Pointers, that cannot be statically analyzed, can be manually resolved through the editor.

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 pointer recoding feasible and enables it to be useful on real-life embedded source codes. Our pointer recoder was effective in recoding 83% of the pointers in the embedded MyBench benchmark [24].

Quantities	GSM	Fix-Point MP3	Floating-Point MP3
Lines of C code	13K	8.7K	3.6K
Functions in C Model	163	67	30
Behaviors in Spec. Model	70	54	43
Interfering pointers	17	23	16
Pointers recoded	17	22	14
Automatic Re-coding time	$\approx$ 1.5 min	$\approx$ 1.5 min	$\approx$ 1 min
Estimated Manual time	170 mins	220 mins	140 mins
Productivity factor	**113x**	**146x**	**140x**

Table 4: Productivity gains for pointer recoding [5].

To estimate the productivity gains, we applied the source recoder to two MP3 audio decoders and a GSM

vocoder application, each spanning thousands of lines of code. From these C codes, we created a model in SLDL suitable for design exploration and synthesis. The design tools successfully performed design space exploration by mapping code and data partitions in the model to different processors and memories. This required that the input model is free of pointers.

Using our source recoder, the pointers were eliminated in a matter of minutes, as shown in Table 4. In the absence of our pointer recoder, the designer must perform the recoding steps manually. The manual time shown in Table 4 is estimated using an average time of 10 minutes per pointer recoding. Clearly, using our pointer recoding approach results in large productivity gains.

5 Conclusion

In this report, we address the critical problem of modeling and parallelization of embedded systems. We have described a *re-coding methodology*. We automate the process of modeling by use of *computer-aided re-coding*. This allows to derive a flexible parallel system model straight from available reference code and thereby drastically shorten the design time.

Our preliminary experiments support the expectation of high productivity gains. Our combination of automatic transformations and interactive control carries potential not only for MPSoC architectures, but also toward general-purpose multi- and many-core processing.

Using designer-controlled computer-aided re-coding, complex model transformations can be realized instantly by a click of button. Thus, tedious coding tasks like exposing communication can be performed in the order of seconds, instead of hours. Moreover, for code transformations of higher complexity, such as converting longer sections of reference code automatically into flexible and parallel design models, even higher productivity gains are expected.

In the long run, our re-coding technique may even have potential to solve the parallel programming problem. Starting from available reference code, the combination of automatic transformations and designer control promises to quickly generate a flexible system model suitable for general parallel architectures. This not only includes MPSoC platforms, but also general-purpose multi- and many-core processors.

References

[1] Jasmin Blanchette and Mark Summerfield. *C++ GUI Programming with Qt 3*. Prentice Hall, February 2004.

[2] Pramod Chandraiah and Rainer Dömer. Specification and design of an MP3 audio decoder. Technical Report CECS-TR-05-04, Center for Embedded Computer Systems, University of California, Irvine, May 2005.

[3] Pramod Chandraiah and Rainer Dömer. An Interactive Model Re-Coder for Efficient SoC Specification. In Achim Rettberg, Mauro C. Zanella, Rainer Dömer, Andreas Gerstlauer, and Franz J. Rammig, editors, *Embedded System Design: Topics, Techniques and Trends*, Boston, MA, 2007. Springer.

[4] Pramod Chandraiah and Rainer Dömer. Designer-Controlled Generation of Parallel and Flexible Heterogeneous MPSoC Specification. In *Proceedings of the Design Automation Conference (DAC)*, June 2007.

[5] Pramod Chandraiah and Rainer Dömer. Pointer re-coding for creating definitive MPSoC models. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*, Salzburg, Austria, September 2007.

[6] Pramod Chandraiah and Rainer Dömer. 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, January 2008.

[7] Pramod Chandraiah and Rainer Dömer. Code and Data Structure Partitioning for Parallel and Flexible MPSoC Specification Using Designer-Controlled Re-Coding. *IEEE Transactions on Computer-Aided Design of Intergrated Circuits and Systems (TCAD)*, 27(6):1078–1090, June 2008.

[8] Pramod Chandraiah, Junyu Peng, and Rainer Dömer. Creating Explicit Communication in SoC Models Using Interactive Re-Coding. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*, Yokohama, Japan, January 2007.

[9] Pramod Chandraiah, Hans Gunar Schirner, Nirupama Srinivas, and Rainer Dömer. System-On Chip Modeling and Design: A case study on MP3 Decoder. Technical Report CECS-TR-04-17, Center for Embedded Computer Systems, University of California, Irvine, July 2004.

[10] Rainer Dömer. The SpecC internal representation. Technical report, Information and Computer Science, University of California, Irvine, January 1999. SpecC V 2.0.3.

[11] Rainer Dömer, Andreas Gerstlauer, and Daniel Gajski. *SpecC Language Reference Manual, Version 2.0*. SpecC Technology Open Consortium, http://www.specc.org, December 2002.

[12] Rainer Dömer, Andreas Gerstlauer, Junyu Peng, Dongwan Shin, Lukai Cai, Haobo Yu, Samar Abdi, and Daniel Gajski. System-on-Chip Environment: A SpecC-based Framework for Heterogeneous MPSoC Design. *EURASIP Journal on Embedded Systems*, 2008(647953):13, July 2008.

[13] Martin Fowler. Refactoring: Improving the design of existing code. In *Proceedings of the Second XP Universe and First Agile Universe Conference on Extreme Programming and Agile Methods - XP/Agile Universe 2002*, page 256, London, UK, 2002. Springer-Verlag.

[14] Daniel D. Gajski, Jianwen Zhu, Rainer Dömer, Andreas Gerstlauer, and Shuqing Zhao. *SpecC: Specification Language and Design Methodology*. Kluwer Academic Publishers, 2000.

[15] Andreas Gerstlauer, Shuqing Zhao, Daniel D. Gajski, and Arkady M. Horak. Design of a GSM vocoder using SpecC methodology. Technical Report ICS-TR-99-11, Information and Computer Science, University of California, Irvine, March 1999.

[16] Thorsten Grötker, Stan Liao, Grant Martin, and Stuart Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[17] Sumit Gupta, Rajesh Kumar Gupta, Nikil D. Dutt, and Alexandru Nicolau. Coordinated parallelizing compiler optimizations and high-level synthesis. *ACM Trans. Des. Autom. Electron. Syst.*, 9(4):441–470, 2004.

[18] Mary W. Hall, Jennifer-Ann M. Anderson, Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, Edouard Bugnion, and Monica S. Lam. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, 29(12):84–89, 1996.

[19] Seema Hiranandani, Ken Kennedy, Chau-Wen Tseng, and Scott K. Warren. The D editor: a new interactive parallel programming tool. In *Supercomputing*, pages 733–742, 1994.

[20] Trolltech Inc. Qt application development framework. http://www.trolltech.com/products/qt/.

[21] K. Kennedy, K. S. McKinley, and C.-W. Tseng. Analysis and transformation in the ParaScope Editor. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, Germany, 1991.

[22] Shih-Wei Liao, Amer Diwan, Robert P. Bosch Jr., Anwar M. Ghuloum, and Monica S. Lam. SUIF explorer: An interactive and interprocedural parallelizer. In *Principles Practice of Parallel Programming*, pages 37–48, 1999.

[23] Eclipse java development tool-kit. http://eclipse.org/jdt/index.html.

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

[25] Microsoft visual studio. http://msdn.microsoft.com/vstudio/.

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

[27] Alberto Sangiovanni-Vincentelli and Grant Martin. Platform-Based Design and Software Design Methodology for Embedded Systems. *IEEE Design and Test of Computers*, 18(6):23–33, 2001.

[28] Alberto L. Sangiovanni-Vincentelli. Quo Vadis SLD: Reasoning about Trends and Challenges of System-Level Design. *Proceedings of the IEEE*, 95(3):467–506, March 2007.

[29] Scintilla source code editing component. http://www.scintilla.org.

[30] SEMATECH Inc. International technology roadmap for semiconductors (ITRS), 2004 update, design. http://www.itrs.net/, 2004.

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

[32] Ines Viskic and Rainer Dömer. A Flexible, Syntax Independent Representation (SIR) for System Level Design Models. In *Proceedings of the EuroMicro Conference on Digital System Design*, Dubrovnik, Croatia, August 2006.