# An Eclipse-based Software Platform for the Recoding Integrated Development Environment (RIDE)

Bin Zhang, Rainer Dömer

Technical Report CECS-09-06
May 26, 2009

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-8059

{binz, doemer}@uci.edu
http://www.cecs.uci.edu/

# An Eclipse-based Software Platform for the Recoding Integrated Development Environment (RIDE)

Bin Zhang, Rainer Dömer

Technical Report CECS-09-06
May 26, 2009

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-8059

{binz, doemer}@uci.edu
http://www.cecs.uci.edu

**Abstract**

*With the increasing abstraction level of embedded system models, System Level Description Languages(SLDL) ,like SystemC or SpeC, have been increasingly used as the design entrance. Having a well described system level model, tools can gradually synthesize them into lower level until the final netlist implementation. However, accompanying with the increase of design ability, new bottlenecks are generated. Many of the system level models are recoded from legacy C reference code, and most of the recoding is nowadays done manually by designers. Due to tedious work, this process is error-prone, time-consuming, and the quality of the final system level model is unstable. This problem has been addressed by a designer-controlled Recoding Integrated Development Environment (RIDE). With both a textual and a graphic editor and a set of analysis and transformation tools, the designer can re-code C reference code into a system level model more efficiently.*

*In this report, we describe an Eclipse-based software platform for RIDE which can serve as a robust and extensible framework for RIDE software development and implementation. We describe in detail the initial software package installation and outline paths for its extension and customization. As one example, we describe a syntax-highlighting feature for the SpecC SLDL.*

# Contents

## List of Figures

# List of Tables

# An Eclipse-based Software Platform for the Recoding Integrated Development Environment (RIDE)

**Bin Zhang, Rainer Dömer**

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA

{binz, doemer}@uci.edu
http://www.cecs.uci.edu

## Abstract

*With the increasing abstraction level of embedded system models, System Level Description Languages(SLDL) ,like SystemC or SpeC, have been increasingly used as the design entrance. Having a well described system level model, tools can gradually synthesize them into lower level until the final netlist implementation. However, accompanying with the increase of design ability, new bottlenecks are generated. Many of the system level models are recoded from legacy C reference code, and most of the recoding is nowadays done manually by designers. Due to tedious work, this process is error-prone, time-consuming, and the quality of the final system level model is unstable. This problem has been addressed by a designer-controlled Recoding Integrated Development Environment (RIDE). With both a textual and a graphic editor and a set of analysis and transformation tools, the designer can re-code C reference code into a system level model more efficiently.*

*In this report, we describe an Eclipse-based software platform for RIDE which can serve as a robust and extensible framework for RIDE software development and implementation. We describe in detail the initial software package installation and outline paths for its extension and customization. As one example, we describe a syntax-highlighting feature for the SpecC SLDL.*

# 1 Introduction

During the past decades years, the technology of Integrated Circuit design has been developed dramatically. Unlike in the early days, when there are only dozens of transistors integrated, a modern chip can contain millions of transistors and the number is doubling every 18 months according to Moore's Law. Pre-designed IPs, like processors or buses, as well as specific hardwares can be integrated into one chip, while even multi-cores & network on chip(NOC) are possible in certain situations.

This is a great opportunity because the high density of integration gives the possibility of implementing more complex systems (like: embedded system) in a very small space. A lot of components, which are used to be implemented mechanically, are nowadays replaced by electronic counterparts.

Meanwhile, it is also a great challenge, because complexity is increasing much faster than the design ability of the current methodology. A new methodology is neccessary to fill the gap between the exponentially increasing complexity and only linearly increasing design ability.

## 1.1 Methodology

To break down the complexity, *Divide and Conquer* is often a good methodology. By partitioning the intangible big problem into tangible small problems level by level, the structure is introduced gradually, and the internal relationships become more clear. Finally, all solvable or implementable "leaf problems", combined with the hierarchy information, form up the solution of the initial complex problem.

This method has been used in hardware design for a long time. In 1960s and 1970s, the hardware designer only have to deal with hundreds of transistors. The system is easily manageable, and transistors can be manipulated directly. In the 1980s, with more and more transistors are integrated onto the chip, the gate level abstraction becomes popular, because with the increasing complexity, the direct transistor design is too hard and costly to be applied. When it comes to the 1990s, RTL modeling starts to appear. The gates are replaced by registers with arithmetic operation filled between them. Along with Verilog and VHDL, more and more designers describe the system at the Register Transfer Level, and automatic synthesizing tools (like Design Compiler) can synthesize them into gate level, then transistor level. When it comes to the 2000s,Transaction Level Modelling(TLM) or System Level Modeling becomes the highest level abstraction.

## 1.2 System Level Description Language

SpecC[13] is the first Transaction Level Modeling language. By introducing the concept of channels, the communication and computation are separated, and communication is described explicitly. The system is abstracted into computations and the explicit transactions between them.

After the system specification is captured in SpecC, the SCE environment[21] can refine this specification into software and hardware(in RTL), and the system can be implemented by the lower level tool chains, like Design Compiler.

Another System Level Description Language is SystemC[15]. The concept shares a lot with SpecC, however, instead of a totally new language, SystemC implements these concepts by using object-oriented libraries in C++.

## 1.3 Hardware Software Co-Design

So far, the System Level Description Language appears to be a good solution to the increasing complexity of system design. However, when the abstraction comes into the system level, the boundary between software and hardware becomes more and more ambiguous. Both SpecC and SystemC are heavily related with C language, which is the most popular software programming language in the embedded system design. When it comes to this point, idea is to implement the hardware and software both in System Level Description Language, which is called System Specification or System Level Model. The System Specification focuses more on the generic computation nature of the application rather than the detailed software or handware implementation,and this model can be later on refined into software and hardware implementation.

## 1.4 C to SLDL Recoder

The System Specification is usually not created from scratch. Many applications, which are used to run on general purpose computation platforms (like PCs), are a good starting point to create the System Specification in SLDL. .

However, because the original C reference code is purely sequential, huge a mount of *”structure information‘‘* needs to be added to the C code. Manually adding this information is tedious, error-prone and time-consuming, which makes this process the new bottleneck.

This problem is addressed by a designer-controlled Recoding Integrated Development Environment (RIDE). With both a textual and a graphical editor and plenty of analysis and transformation tools, the designer can re-code the C reference code into the System Level Model more efficiently.

We describe the motivation for the recoding idea in Section 2, discuss related work in Section 3, and then outline the new Recoding Integrated Design Environment (RIDE) in Section 4. Specific design and implementation details of an Eclipse-based software framework for future RIDE development are described in Section 5 (and the Appendix) of this report.

## 2 Motivation

During the past few years, the bottleneck of system design has shifted from the RTL Model creation to the Specification Model. Before that, the bottleneck of the design is the RTL coding. Dozens of engineers spent several months to write the RTL model of a design according to the specification. After the coding is finished, the RTL models are verified, simulated and finally synthesized into netlist by some commercial synthesizer (like, Design Compiler). With appearance of SLDL, like SpecC or SystemC, people start to write the system level models from scratch or by modifying reference C code. The SpecC or SystemC model are then automatically synthesized into the RTL model, and further netlist model.

However, the creation of SpecC or SystemC model is not an easy job. Due to immaturity of this process, many problems exist. In the following part, we define these problems from three perspectives.

### 2.1 Time Issue

According to our experience, manually re-coding the reference implementation of MP3 decoder into a mature SpecC or SystemC model takes 12-14 weeks (shown in Fig.1 [2]). In contrast, the following processes from Specification Model to Implementation Model only takes less than 1 week.

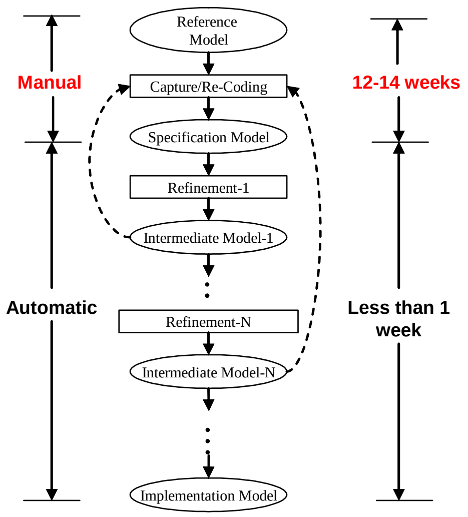


Figure 1: MP3 Decoder Design Time

### 2.2 Quality Issue

An error costs 10 times more energy and money to find it in the lower level than in the higher level. This means if a system level model error is found in the netlist level model, it will cost us 1000 times more money and energy than finding the error in the system level model. So, an error-free model at the system level is very valuable.

While at the same time, most of the system model is created by modifying the reference code. This purely manual process is error-prone.

## 2.3 Cost Issue

Currently the payment to the engineer is often the lion's share of a project. One engineer costs about 80,000 dollars per year. If most of their time is spent on the tedious works which can be automated, that is actually a great waste of human resource.

In summary, the **Time**, **Quality** and **Cost** issues make the system level modeling a major bottleneck of the whole system design process. As depicted in Fig.2, how to bridge the gap in shorter time with less spending and higher quality has become the new challenge of embedded system design. These problems will be addressed in the following sections.

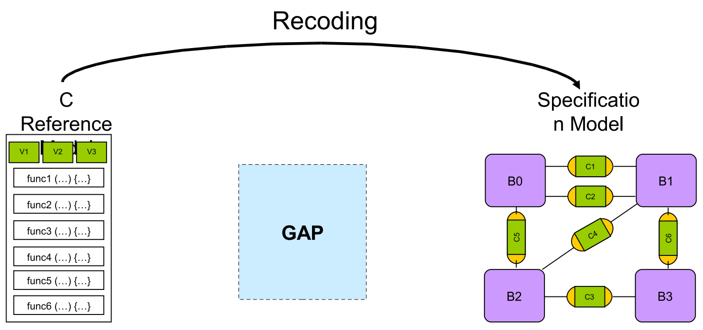


Figure 2: Recoding Gap

# 3 Related Work

Until today, there are actually only few papers to address this problem of creating the system level specification models. In this section, we will list the projects that are related to this process.

## 3.1 Related Work Done in Other Teams

**MAPS:** MAPS[16] is project undertaken at RWTH Aachen University. It proposed an integrated framework to parallelize the C applications for MPSoC platforms. The description of the target MPSoC is given as an input. It extracts coarse-grained parallelism, and a set of tools have been developed for the framework. The structure of MAPS is depicted in Fig.3.

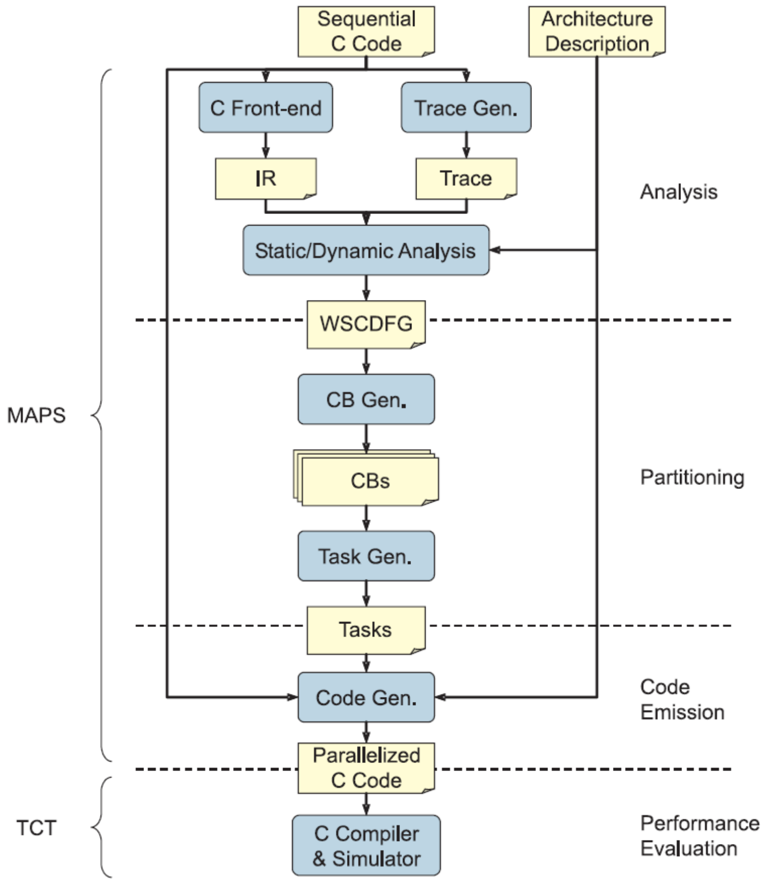


Figure 3: MAPS Structure [16]

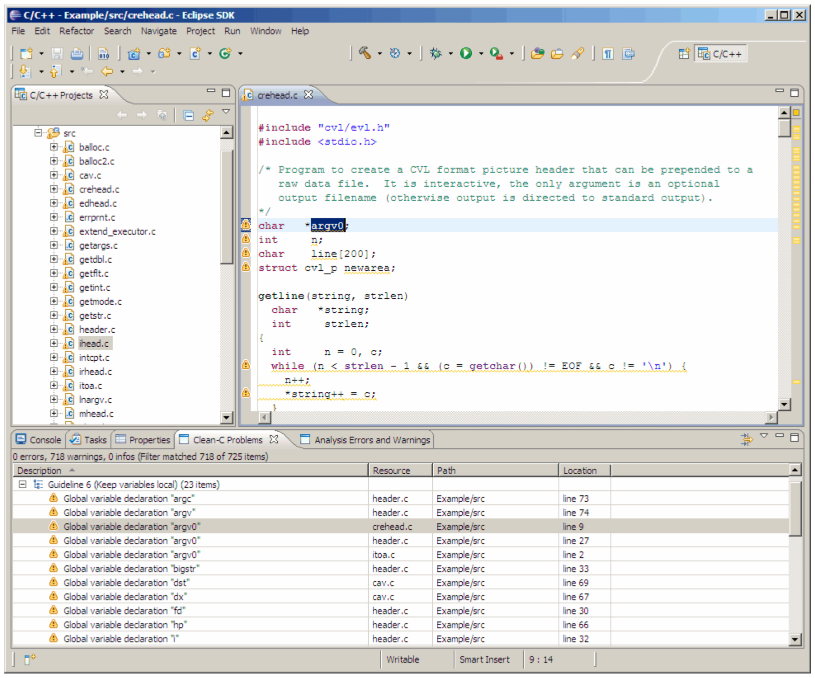
However, this work focuses on modifying the code to creating a parallel program on MPSoC rather than creating the specification model of the application. That means MAPS is mostly a software flow with small code in the future synthesized into hardware. However, this feature is very important because most of the MPSoC chips are customized, and part of the HW are ASICs instead of purely general on-shelf processors.

Meanwhile, this work is based on a so-called Tightly-Coupled-Thread(TCT) programming model[20] rather than some mature system level modeling model, like SpecC or SystemC.

**CleanC:** Another project that also addresses this problem is called CleanC [8]. It can analyze the C code for a variety of coding patterns that make the C code hard to understand by humans and difficult to analyze by tools, like:

- Distinguish source file from header file
- Use macros for constants and conditional exclusion
- Keep variables local
- Make sure a pointer points to only one data set
- Do not use recursive function calls
- Do not use functions with varargs
- Use switch statements instead of function pointers
- Use the manifest loop pattern
- Make the control ow regular
- Keep side-effects out of expressions
- Use indexes instead of pointer arithmetic
- Do not cast to / from a pointer
- Cast the result of malloc() to the correct type
- Use arithmetic operators to perform calculations
- Avoid the dark corners of the C standard
- Respect the semantics of types

This tool is implemented as the plugin in Elicpse development environment [11] with CDT [1] installed. The violations are listed under the ”Clean-C Problem” tab as well as in the code. A snapshot of the tools is given in Fig.4.3



```
#include "cvl/evl.h"
#include <stdio.h>

/* Program to create a CVL format picture header that can be prepended to a
   raw data file.  It is interactive, the only argument is an optional
   output filename (otherwise output is directed to standard output).
*/
char    *argv0;
int     n;
char    line[200];
struct cvl_p newarea;

getline(string, strlen)
  char    *string;
  int     strlen;
{
  int     n = 0, c;
  while (n < strlen - 1 && (c = getchar()) != EOF && c != '\n') {
    n++;
    *string++ = c;
  }
```

0 errors, 718 warnings, 0 infos (Filter matched 718 of 725 items)

| Description | Resource | Path | Location |
|---|---|---|---|
| Guideline 6 (Keep variables local) (23 items) | | | |
| Global variable declaration "argc" | header.c | Example/src | line 73 |
| Global variable declaration "argv" | header.c | Example/src | line 74 |
| Global variable declaration "argv0" | crehead.c | Example/src | line 9 |
| Global variable declaration "argv0" | header.c | Example/src | line 27 |
| Global variable declaration "argv0" | itoa.c | Example/src | line 2 |
| Global variable declaration "bigstr" | header.c | Example/src | line 33 |
| Global variable declaration "dst" | cav.c | Example/src | line 69 |
| Global variable declaration "dx" | cav.c | Example/src | line 67 |
| Global variable declaration "fd" | header.c | Example/src | line 30 |
| Global variable declaration "hp" | header.c | Example/src | line 66 |
| Global variable declaration "i" | header.c | Example/src | line 32 |

Figure 4: Snapshort of CleanC

However, similar to the MAPS, CleanC is also only a pure software flow. It can only analyze a handful features and all the code transformation work has to be done by the designer.

**C-to-SystemC Synthesizer**: A C-to-SystemC Synthesizer is mentioned in paper[25] with structure shown in Fig.5 It uses some scheduling methods to generate SystemC code from C code. However, given complexity of some reference C code, this method appears quite simple to work.

**Others:** Other related work includes parallelizing compilers and a software engineering technique known as refactoring. Intensive research has been done in the parallel computing field to automatically parallelize the software, such as interprocedural analysis[18], symbolic analysis[19], and loop transformation[18] [7]. However, these are more focused on general computing or scientific computing, which has little touch in the embedded system community. For the code refactoring[17] [22], the purpose is different from our recoding. Refactoring is more focused on improving the human readability, understandability, and maintainability of the source code, while our recoding is focused on creating the system level model by exposing the communication and concurrency.

## 3.2 Related Work Done in Our Team

Besides the related work done in other teams, this topic has been heavily researched by Pramod Chandraiah. Several papers [3], [4], [5], [6] have been published on this recoding process, and a

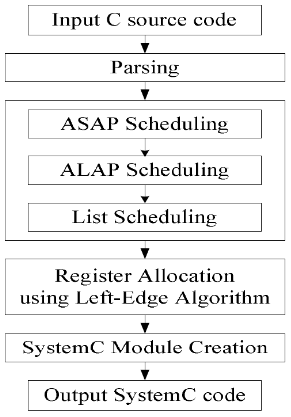


Figure 5: Structure of C to SystemC Synthesizer

designer controlled recoder [2] has been created.

### 3.2.1 Recoding Flow

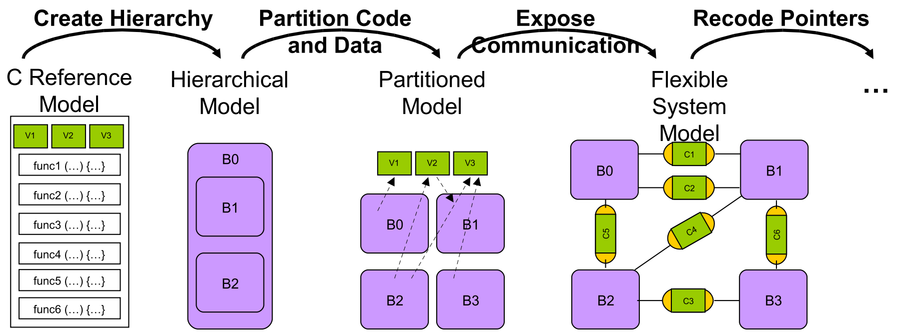


Figure 6: Basic Recoding Process

The basic re-coding process by P.Chandraiah is depicted in Fig.6. As shown, it takes 4 major steps to start from a pure C Reference Model to a Flexible System Model without unnecessary pointers:

- **Creating Structural Hierarchy**: This process encapsulates the computation. The original functions in the C Reference Model are packed hierarchily into **behaviors**.
- **Code & Data Partitioning**: In this process, the C Reference Model structure will be dismantled. Functions and global variables will be partitioned into reasonable pieces to give

more flexibility for the later process. Several typical transformations include: loop splitting, cumulative access type analysis, partition of vector dependents and synchronizing dependent variables.

- **Creating Explicit Communication**: After the code has been divided into small pieces, the variables are replaced by channels to connect different behaviors to compose the whole system. The concept of channel is the most significant contribution of system level modeling. They contain more information than variables, and these information can be used, later on, by compiler or synthesizer to generate lower level representation.
- **Recode Pointers**: After we get the flexible system model, the pointer recoding will be applied. Pointer is an very common concept in software but sometimes abused in the code. In the pointer recoding process, these unnecessary pointers are removed.

Inside these steps, detailed re-coding techniques are:

- Loop Splitting
- Cumulative Access-Type Analysis
- Partitioning Vector Dependants
- Breaking Composite Structures
- Synchronizing Dependent Variables
- Variable Rescoping

### 3.2.2 CUTE

Based on the concept mentioned above, an Interactive Source Recoder has been implemented. Working as an union of editor, abstract syntax tree(AST), parser, transformationer and code generator, many of the transformations can be done by justing clicking one button. The structure of CUTE is depicted in Fig.7

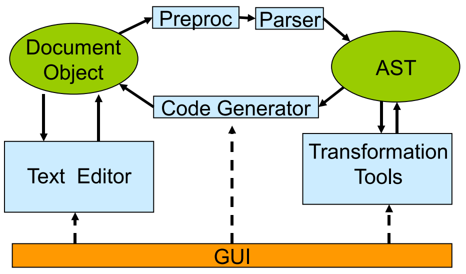


Figure 7: CUTE structure

## 4 Recoding Integrated Development Environment (RIDE)

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.

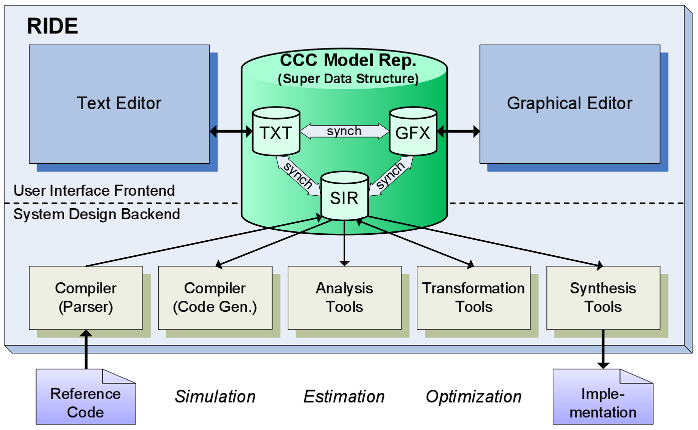


Figure 8: Recoding Integrated Development Environment.

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 abstraction. It can be used to re-code intermediate system design models, as well as the initial C reference implementation in order to produce an optimal system model.

The conceptual structure of the proposed Recoding IDE is shown in Figure 8. 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. The RIDE is build on the eclipse [11].

### 4.1 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 [15] and SpecC [10] SLDLs.

The graphical editor is envisioned to present a hierarchical diagram of the design model that can be used for visualization, navigation, and modification operations. Supported visualization operations include zoomin/zoomout, 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.

All 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 4.2 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.

### 4.2 RIDE Super Data Structure

At the core of the Recoding IDE, we plan a complex data structure that 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). We plan to support both SystemC [15] and SpecC [10] 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.

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.

The **syntax-independent representation** (SIR) [24, 9] 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, and optimization. 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(including the MDS), are combined into a RIDE super data structure that will keep the design model accurately reflected and updated in each representation. Synchronization functions (*synch* in Figure 8) 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 is a key contribution 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 have seen in [2], an early prototype implementation of two synchronized data structures showed sufficient responsiveness, even though a brute-force file interface was used for the synchronization operations.

## 4.3 RIDE Backend

The RIDE backend is planned 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.

We would like to emphasize that the strength of RIDE lies in this system design backend. In other words, the analysis and transformation tools build the core of the re-coding environment.

To provide an overview about expected analysis and transformation tasks, we can conceptually categorize our 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.

## 5 Implementation Details

We use the Eclipse framework [11] to implement RIDE. Eclipse is a software platform to provide a basement for integrated development environment (IDE) development. It is written in Java, and IDE developers can extend its functions by installing plugins written for it.

Eclipse itself is not a single monolithic program.The basement is called the runtime system, which is based on Equinox[12]. The runtime system is very light weight, and all the other functions are implemented as plugins on top of it. Each plugin can use the functions of another plugin, and also provide functions which can be used by other plugins.

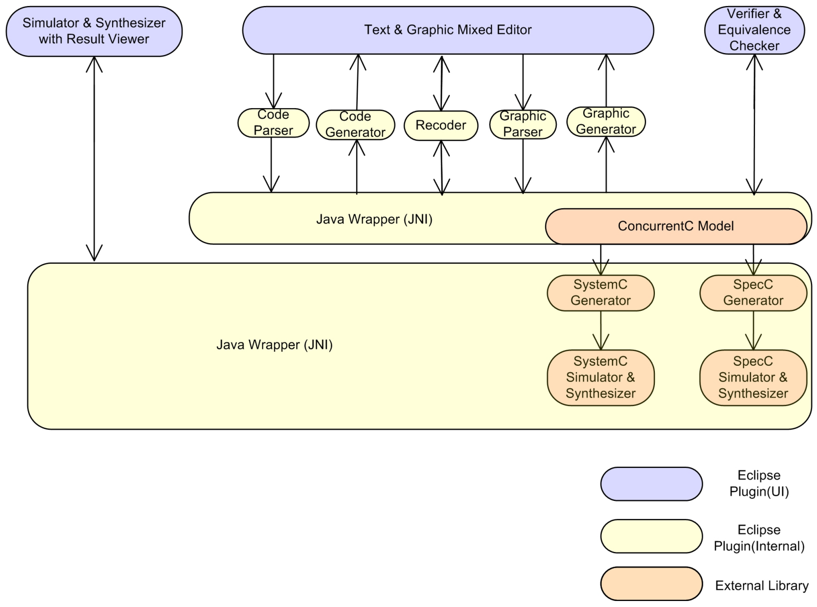


Figure 9: Internal Software Architecture of RIDE

The software architecture of RIDE is depicted in Fig.9. As shown, the RIDE frontend (4.1) is implemented as a plugin in the Eclipse environment. The editor can be derived from the editor offered in Eclipse, while other components need to be written from scratch.

For the RIDE super data structure (4.2), we would like to reuse the repository data structure SIR. However, due to that the SIR was originally written in C++, we have to create a wrapper for this data structure. The tool we use to automaticly generate this wrapper is called SWIG [23]. The wrapper generation flow is depicted in Fig.10. After the wrapper is generated, the SIR functions can be used in Eclipse by calling the Java interface.

For the RIDE backend (4.3), some of the tools are implemented in the same way as the super data structure. The repository code in C++ can be reused, and the wrapper in Java is generated to make it possible be called in the Eclipse plugin.

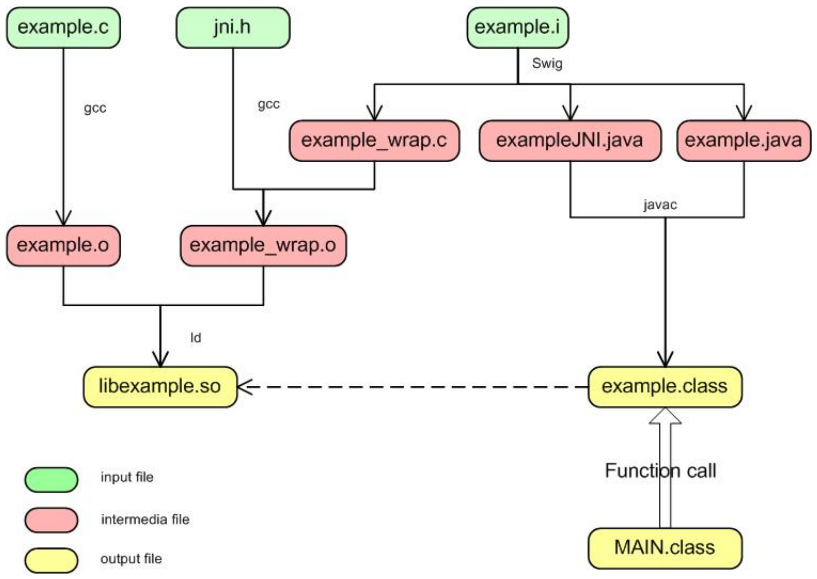


Figure 10: The SWIG wrapper generation flow

For some other tools with no good repository code, there are two choices: first is write the new tools in Java. This is an easier way, but the consistency is broken, because part of the tools are Java while part of the tools are in C++. The second choice is still use C++ to write these tools and use SWIG to create a wrapper for them. This method can keep the consistency of the project, but takes longer time. And the cross-platform feature of Java cannot be used in this case, because the code in C++ is not cross-platform.

For a more detailed documentation of the implementation of the RIDE software platform, please refer to the Appendix.

## 6 Conclusion

In this report, we have addressed the gap between C reference code and SLDL. New tool structures are proposed with the consideration of time, quality and cost. With both text and graphic interface in the proposed RIDE framework, the flexibility of the design entrance is increased. The designer can apply the transformations in either text form or graphic. New transformation tools can be offered with higher quality code generated. New analysis tools can also facilitate the understanding of the model.

Similar to the RTL languages 10 years ago, SLDL will be the standard in the future. More applications will be designed from SLDL, and more mature tools will come out. In the long term, the design of embedded systems may be so simple that even the initial algorithm designer can implement their own system from pure C code without having any knowledge of the hardware details. However, this progress do not happen automatically without EDA engineer's work.

## 7 Acknowledgement

This work has been supported in part by the National Science Foundation under grant NSF CAREER Award #0747523. The authors thank the NSF for supporting this research.

## References

[1] CDT Project. http://www.eclipse.org/cdt/.

[2] Pramod Chandraiah and Rainer Dömer. An Interactive Model Re-Coder for Efficient SoC Specification. In *Proc. International Embedded System Symposium, Embedded System Design: Topics, Techniques and Trends*, *Springer*, Irvine, California, USA, May 2002.

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

[4] Pramod Chandraiah and Rainer Dömer. Pointer Re-coding for Creating Definitive MPSoC Models. In *CODES-ISSS*, Salzburg, Austria, September 2007.

[5] 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, South Korea, January 2008.

[6] Pramod Chandraiah and Rainer Dömer. Code and Data Structure Partitioning for Parallel and Flexible MPSoC Specification Using Designer-Controlled Re-Coding. In *IEEE Tran. on Computer-Aided Design of Inegrated Circuits and System*, June 2008.

[7] J.-H Chow, L.E.Lyon, and V.Sarkar. Automatic parallelization for symmetric shared-memory multiprocessors. In *Proc. Conf. CASCON*, 1995.

[8] CleanC home page. http://www.imec.be/CleanC/.

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

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

[11] Eclipse Project. http://www.eclipse.org.

[12] Equinox Project. http://www.eclipse.org/equinox/.

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

[14] Andreas Gerstlauer, Rainer Dömer, Junyu Peng, and Daniel D. Gajski. *System Design: A Practical Guide with SpecC*. Kluwer Academic Publishers, 2001.

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

[16] J.Ceng, J.Castrillon, W.Sheng, H.Scharwächter, R.Leupers, and H.Meyr G.Ascheid. MAPS: An Integrated Framework for MPSoC Application Paralleization. In *Proceedings of the Design Automation Conference (DAC)*, Anaheim, California,USA, June 2008.

[17] M.Fowler. Refactoring: Improving the design of existing code. In *Proc. 2nd XP Universe, 1st Agile Universe Conf. Extreme Program. Agile Methods-XP/Agile Universe*, 2002.

[18] M.H.Hall, S.P.Amarasinghe, B.R.Murphy, S.W.Liao, and M.S.Lam. Detecting coarse-grain parallelism using an interprocedural paralleizing compiler. In *ACM/IEEE Conf.Supercomputing*, 1995.

[19] M.R.Haghighat and C.D.Polychronopouls. Symbolic analysis for parallelizing compilers. In *ACM Trans. Program. Lang. Syst.*, July 1996.

[20] M.Z.Urfianto, T.Isshiki, A.U.Khan, D.Li, , and H.Kunieda. A Multiprocessor System-on-Chip Architecture with Enhanced Compiler Support and Efficient Interconnect. In *IP-SOC 2006,Design and Reuse, S.A.*, 2006.

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

[22] Index of Refactorings. thhp://www.refactoring.com/catalog/index.html.

[23] SWIG Project. http://www.swig.org/.

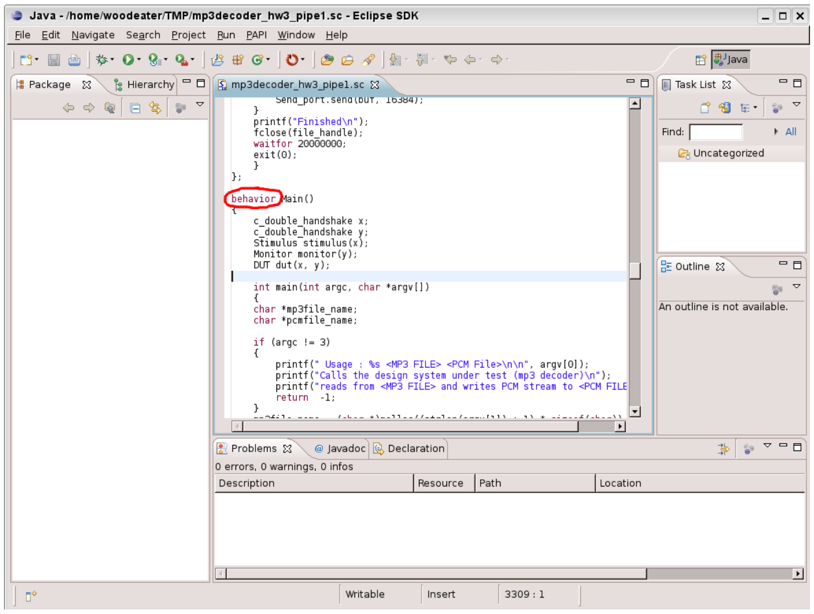
[24] Ines Viskic and Rainer Dömer. A Flexible, Syntax Independent Representation (SIR) for System Level Design Models. In *EURODSD*, Dubrovnik, Croatia, Aug 2006.

[25] Myoung-Keun You and Gi-Yong Song. Implementation of a C-to-SystemC synthesizer prototype. In *ASICON*, October 2007.

## A RIDE Implementation

### A.1 Introduction

As depicted in Fig.9, we have implemented an Eclipse-based text editor with syntax-highlighting. A snapshot of this editor is shown in Fig.11. This editor derives from the Eclipse editor with syntax highlight function added. It uses a scanner to scan all the keywords of SpecC in the file and uses color to print them in the editor.



```
        Send_port.send(buf, 16384);
    }
    printf("Finished\n");
    fclose(file_handle);
    waitfor 20000000;
    exit(0);
    }
};

behavior Main()
{
    c_double_handshake x;
    c_double_handshake y;
    Stimulus stimulus(x);
    Monitor monitor(y);
    DUT dut(x, y);

    int main(int argc, char *argv[])
    {
    char *mp3file_name;
    char *pcmfile_name;

    if (argc != 3)
    {
        printf(" Usage : %s <MP3 FILE> <PCM File>\n\n", argv[0]);
        printf("Calls the design system under test (mp3 decoder)\n");
        printf("reads from <MP3 FILE> and writes PCM stream to <PCM FILE
        return  -1;
    }
```

An outline is not available.

0 errors, 0 warnings, 0 infos

| Description | Resource | Path | Location |
|---|---|---|---|

Figure 11: RIDE editor

### A.2 Environment Setup

#### A.2.1 Eclipse 3.3 Setup

Step 1: Download the Eclipse 3.3 from the following website:
http://archive.eclipse.org/eclipse/downloads/

Step 2: Extract the tar ball by the following command:

```
gunzip < eclipse-SDK-3.3-linux-gtk.tar.gz | tar xvf -
```

Step 3: Eclipse is ready to be used in the eclipse folder (No other plug-in needed)

### A.2.2 Java Runtime Enviroment Setup

Step 1: Download the JRE 1.6 from the following website:
http://java.sun.com/javase/downloads/index.jsp

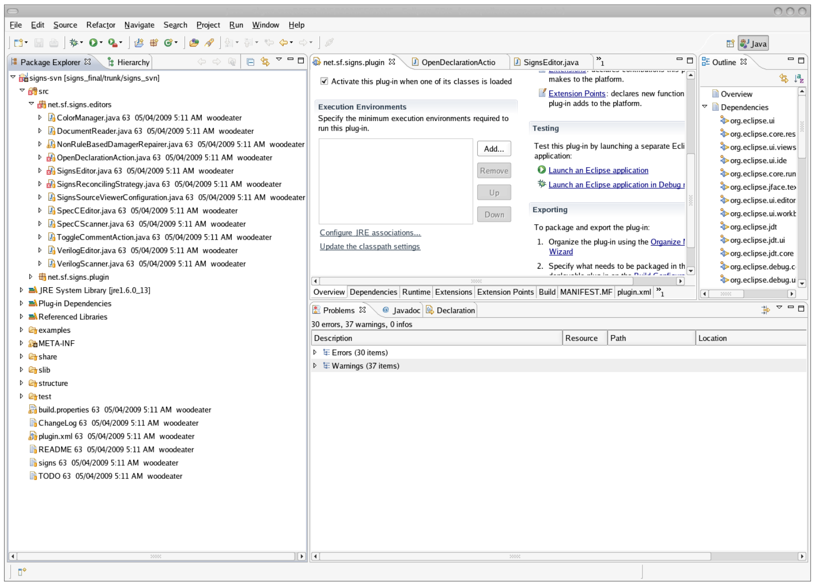
Step 2: Install the JRE by the following command:

```
./jre-6y13-linux-i586.bin
```

## A.3 Project Description

After the enviroment setup, we can build our Eclipse plug-in. The source code can be found in the subfolder **workspace**. Currently, only the syntax highlighting is working in the project.

This project is developed in the Eclipse 3.3. After the project is opened in the Eclipse enviroment, the file tree can be on the left of the IDE (shown in Figure 12).



```
signs-svn [signs_final/trunk/signs_svn]
  src
    net.sf.signs.editors
      ColorManager.java 63  05/04/2009 5:11 AM  woodeater
      DocumentReader.java 63  05/04/2009 5:11 AM  woodeater
      NonRuleBasedDamagerRepairer.java 63  05/04/2009 5:11 AM  woodeater
      OpenDeclarationAction.java 63  05/04/2009 5:11 AM  woodeater
      SignsEditor.java 63  05/04/2009 5:11 AM  woodeater
      SignsReconcilingStrategy.java 63  05/04/2009 5:11 AM  woodeater
      SignsSourceViewerConfiguration.java 63  05/04/2009 5:11 AM  woodeater
      SpecCEditor.java 63  05/04/2009 5:11 AM  woodeater
      SpecCScanner.java 63  05/04/2009 5:11 AM  woodeater
      ToggleCommentAction.java 63  05/04/2009 5:11 AM  woodeater
      VerilogEditor.java 63  05/04/2009 5:11 AM  woodeater
      VerilogScanner.java 63  05/04/2009 5:11 AM  woodeater
    net.sf.signs.plugin
  JRE System Library [jre1.6.0_13]
  Plug-in Dependencies
  Referenced Libraries
  examples
  META-INF
  share
  slib
  structure
  test
  build.properties 63  05/04/2009 5:11 AM  woodeater
  ChangeLog 63  05/04/2009 5:11 AM  woodeater
  plugin.xml 63  05/04/2009 5:11 AM  woodeater
  README 63  05/04/2009 5:11 AM  woodeater
  signs 63  05/04/2009 5:11 AM  woodeater
  TODO 63  05/04/2009 5:11 AM  woodeater
```

30 errors, 37 warnings, 0 infos

| Description | Resource | Path | Location |
|---|---|---|---|
| Errors (30 items) | | | |
| Warnings (37 items) | | | |

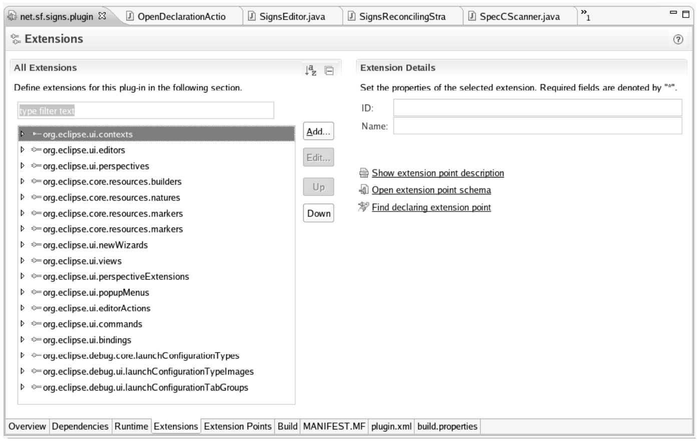
File Edit Source Refactor Navigate Search Project Run Window Help
Package Explorer
Hierarchy
net.sf.signs.plugin
OpenDeclarationActio
SignsEditor.java
Activate this plug-in when one of its classes is loaded
Execution Environments
Specify the minimum execution environments required to run this plug-in.
Add...
Remove
Up
Down
Configure JRE associations...
Update the classpath settings
Extension Points: declares new function plug-in adds to the platform.
Testing
Test this plug-in by launching a separate Eclipse application:
Launch an Eclipse application
Launch an Eclipse application in Debug mode
Exporting
To package and export the plug-in:
1. Organize the plug-in using the Organize Manifests Wizard
2. Specify what needs to be packaged in the
Overview Dependencies Runtime Extensions Extension Points Build MANIFEST.MF plugin.xml
Outline
Overview
Dependencies
org.eclipse.ui
org.eclipse.core.res
org.eclipse.ui.views
org.eclipse.ui.ide
org.eclipse.core.run
org.eclipse.jface.tex
org.eclipse.ui.editor
org.eclipse.ui.workb
org.eclipse.jdt
org.eclipse.jdt.ui
org.eclipse.jdt.core
org.eclipse.debug.c
org.eclipse.debug.ui
Problems
Javadoc
Declaration
Java

Figure 12: Project Overview

As shown, there are 12 classes in this project. The most important class in the project is ”SpecCEditor”. In this class, the highlight keywords are defined.

After the keywords are defined, the next step is to associate the file with ”sc” extension. This association process can be done in the MANIFEST.MF file, Extensions Tab (Figure 13). Find the ”org.eclipse.ui.editors” on the left side, and associate the extension by modifying the ”SpecC Editor” property.

After the project has been properly set up, the project can be tested by clicking the ”Launch an Eclipse application” button in the Overview Tab of MANIFEST.MF file (Figure 14). A new eclipse



Extensions

All Extensions

Define extensions for this plug-in in the following section.

type filter text

- org.eclipse.ui.contexts
- org.eclipse.ui.editors
- org.eclipse.ui.perspectives
- org.eclipse.core.resources.builders
- org.eclipse.core.resources.natures
- org.eclipse.core.resources.markers
- org.eclipse.core.resources.markers
- org.eclipse.ui.newWizards
- org.eclipse.ui.views
- org.eclipse.ui.perspectiveExtensions
- org.eclipse.ui.popupMenus
- org.eclipse.ui.editorActions
- org.eclipse.ui.commands
- org.eclipse.ui.bindings
- org.eclipse.debug.core.launchConfigurationTypes
- org.eclipse.debug.ui.launchConfigurationTypeImages
- org.eclipse.debug.ui.launchConfigurationTabGroups

Extension Details

Set the properties of the selected extension. Required fields are denoted by "*".

ID:

Name:

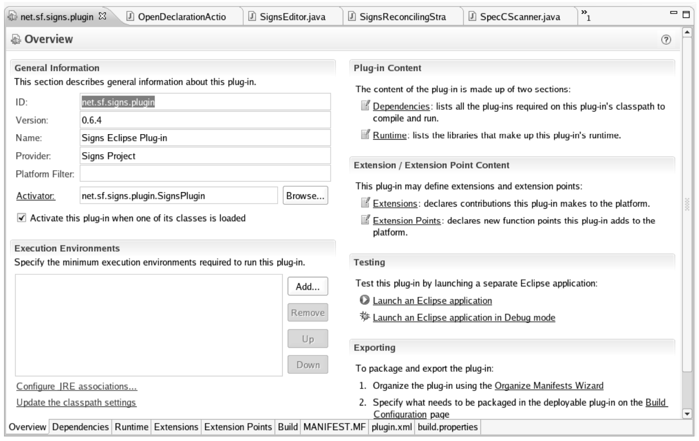
Show extension point description

Open extension point schema

Find declaring extension point

Figure 13: File Extension Association

with the syntax highlight plug-in is opened. We can open a SpecC file with ”sc” extension to test wether the syntax highlight works or not. (Figure 11)



Overview

General Information

This section describes general information about this plug-in.

| | |
|---|---|
| ID: | net.sf.signs.plugin |
| Version: | 0.6.4 |
| Name: | Signs Eclipse Plug-in |
| Provider: | Signs Project |
| Platform Filter: | |
| Activator: | net.sf.signs.plugin.SignsPlugin |

✓ Activate this plug-in when one of its classes is loaded

Execution Environments

Specify the minimum execution environments required to run this plug-in.

Configure JRE associations...

Update the classpath settings

Plug-in Content

The content of the plug-in is made up of two sections:

Dependencies: lists all the plug-ins required on this plug-in's classpath to compile and run.

Runtime: lists the libraries that make up this plug-in's runtime.

Extension / Extension Point Content

This plug-in may define extensions and extension points:

Extensions: declares contributions this plug-in makes to the platform.

Extension Points: declares new function points this plug-in adds to the platform.

Testing

Test this plug-in by launching a separate Eclipse application:

Launch an Eclipse application

Launch an Eclipse application in Debug mode

Exporting

To package and export the plug-in:

1. Organize the plug-in using the Organize Manifests Wizard
2. Specify what needs to be packaged in the deployable plug-in on the Build Configuration page

Figure 14: Project Testing

## B Reference Project

In this design, we use the framework of the Signs project developed by Guenter Bartsch from University of Stuttgart. The information of the Signs project can be found in the website:
http://www.iti.uni-stuttgart.de/~bartscgr/signs/wiki/index.php/Main_Page