Managing Traceability Links between Specifications and Implementations of Hardware/Software Interfaces

Dejun Qian
dejun@cs.pdx.edu

Computer Science Department, Portland State University

Abstract. Traceability links between specifications and implementations of hardware/software (HW/SW) interfaces play a crucial role in improving the quality of computer systems. On one hand, HW/SW interfaces are often quite detailed, on low abstraction levels, thus demanding fine-grained traceability links. On the other hand, HW/SW interface specifications are typically stable while their hardware and software implementations are constantly evolving even after their releases, which requires robust traceability links. In this paper, we present a framework for managing traceability links between specifications and implementations of HW/SW interfaces. This framework supports fine-grained traceability links between basic elements of specifications and constructs of implementation languages. It also supports automatic migration of links as implementations evolve. We have implemented a prototype tool, namely coDoc, in support of this framework. We have applied coDoc to ten versions of three virtual devices in the QEMU virtual machine. Experimental results show that our framework fully supports the types of traceability links between the virtual devices and their specifications, and it accurately migrates valid links from version to version.

1 Introduction

The majority of software engineering work is devoted to maintaining legacy software systems other than developing new ones [1]. It is estimated that up to 60% of software maintenance time is spent on program comprehension [11]. Software developers often find it difficult to extract the knowledge about a software system, because the knowledge is often implicitly expressed, and distributed in a variety of software artifacts. Examples of software artifacts include source code, design documents, application programming interface specifications, bug reports, etc.

To make the knowledge about a software system explicit, reserachers have built various techniques to recover the semantic connections among software artifacts, especially between documentation and source code [2] [7] [13]. These semantic connections are often called *traceability links*. Effectively maintaining the traceability links can help improving the productivity of the developer and the quality of the software [14].

Unfortunately, these techniques are not ready to be applied to implementations and specifications of HW/SW interfaces, even though these implementations are an important category of software that caused most of the system failures in both Windows and Linux operating systems [17] [6], HW/SW interface forms the interaction point of

hardware and software, and defines the protocol for the hardware and software to communicate. The implementation of HW/SW interfaces refers to hardware design, device drivers and virtual device prototypes, who implement the communication logic on both sides of the interfaces. HW/SW interface specifications describe the HW/SW interfaces in natuaral lagnguage, and guide the implementation of the interfaces. HW/SW interface specifications play a crucial role in the comprehension and maintenance of the implementations of the interfaces. The reason why existing traceability link recovering techniques do not work for HW/SW interfaces is that these traceability links need to be fine-grained, which is not supported by the existing techniques. HW/SW interfaces are mainly composed of software-programmable hardware registers mapped to the processor's address space. The low-level abstraction of the software/hardware interface makes the traceability links fine-grained. Traceability links between the specifications and implementations usually connect statements, which operate the registers, or expressions, which represent certain bits of registers, to the register bit descriptions in the specifications. However, previous tools usually maintain tracebility links between the classes or functions of the source code and sections or paragraphs of the documentation, which is more coarse-grained.

We present a framework for managing traceability links between specifications and implementations of HW/SW interfaces. This framework supports fine-grained links between basic elements of specifications and constructs of implementation languages. We design a data model to support the framework, and organize the data for the traceability links. Our framework also supports automatic migration of links as implementations evolve.

We have implemented a prototype tool, namely coDoc, in support of this framework. With this tool, the developer can import and manage implementation source code and interface specifications, select a certain piece of implementation (POI) and a certain piece of specification (POS) of interest, create a traceability link connecting the seleted POI and POS, check out a different version of source code, review and verify the existing traceability links. We applied coDoc to three virtual device prototypes and their specifications, recovered traceability links, and verified the traceability links on other versions of the driver code. Experimental results show that our framework fully supports the types of traceability links between the virtual devices and their specifications, maintain traceability links with required granularity, and it accurately migrates valid links from version to version. It preserves 70% of the existing traceability links when code evolves to a new version.

the data need to be updated

This paper makes the following key contributions:

- Presented a framework for managing traceability links between specifications and implementations of HW/SW interfaces with accuracy, robustness and fine-granularity.
- Developed coDoc, a tool implementing the approach.
- Evaluated our approach with realistic examples.

The rest of the paper is organized as follows. Section 2 examplifies and defines the problem that we address in this paper. Section 3 and section 4 illustrate the design and implementation details of our approach. Section 5 provides some preliminary evaluation results. Section 6 reviews the related work. We conclude our approach and discuss the future work in Section 7.

2 Problem Statement

The interfaces between software and hardware are composed of registers exposed by hardware devices and accessible by software. The implementation of the interfaces should conform to the specification in order to function correctly. Suppose we have to maintain and understand this implementation, we need to establish relationships between the implementation and the specification. Fig. 1(a) presents part of code extracted from a virtual device prototype for MAX7310 in QEMU. A virtual device prototype is a software implementation emulating a hardware device. MAX7310 is a 8-bit I/O port expander. QEMU (Quick EMUlator)[1] is virtual machine used widely in industry to assist hardware related software development. The source code for this implementation is written in C language. From the name of the function, we can guess the function shown is trying to reset the device to its initial state. However, we can not figure out what the values in the right sides of the assignment statements mean only by reading the code here. Fig. 2 includes the information extracted from the specification of MAX7310. In this figure, we can see three tables for three registers: polarity register, direction register and status register. These tables present the bit arrangement and the default value for each register. By looking at this specification, now we can see that the values used in the code are defined in the specification. The code uses the hex format, and the specification use binary format. The piece of code and the piece of specification labeled with the same number are related, which means the piece of specification explains the piece of code with the same label.

Code evolves all the time in its lifecycle. By checking the QEMU repository, we can find that the first version of the code in Fig 1 was created on May 23, 2007, and the last change is on May 1, 2013. This code has changed 21 times till now, and it still keeps changing. Fig. 1(b) shows an example of a new version of the code in Fig. 1(a). Line 13, 25 and 29 of the code in Fig. 1(a) has changed to the line 12, 24 and 28 in Fig. 1(b).

Compared to code, specification is relatively stable. The specification in Fig 2 only has three versions, and the last change happened in 2005. The specification remains unchanged since the code was created.

This example is simple, but reveals three basic requirements for the management of traceability links between specification and implementations as follows.

Requirement 1: Traciability links should be able to uniquely refer to and accurately locate the relevant pieces of code in the source file and the piece of specification in the specification file. As in the example, there should be a way to locate the code pieces and specification pieces surrounded by the boxes labeled by 1, 2 and 3.

Requirement 2: Traceability links should be able to provide referrence at statement level or expression level for source code and at sentence level or word level for specification. As in this example, the code pieces and specification pieces surrounded by the boxes labeled by 1, 2 and 3 are all on this level.

Requirement 3: Traceability links should be immune from the evolusion of unrelated code. We should grarantee that the identifier created for the old version of code works for the new version of code if the code piece refered by the identifier does not change.

[1] `http://www.qemu.org`

```
          ...
13(217): struct max7310_s {
              ...
19(314):      uint8_t direction;
20(337):      uint8_t polarity;
21(359):      uint8_t status;
              ...
25(448): };
26(451):
27(452): void max7310_reset(i2c_slave *i2c)
28(487): {
29(489):      struct max7310_s *s = (struct max7310_s *) i2c;
              ...
31(571):      s->direction = 0xff;   (1)
32(596):      s->polarity = 0xf0;    (2)
33(620):      s->status = 0xff;      (3)
              ...
35(665): }
```

(a) The Old Version

```
          ...
12(201): typedef struct {
              ...
18(296):      uint8_t direction;
19(319):      uint8_t polarity;
20(341):      uint8_t status;
              ...
24(430): } MAX7310State;
25(446):
26(447): void max7310_reset(i2c_slave *i2c)
27(482): {
28(484):      MAX7310State *s = (MAX7310State *) i2c;
              ...
30(558):      s->direction = 0xff;   (1)
31(583):      s->polarity = 0xf0;    (2)
32(607):      s->status = 0xff;      (3)
              ...
34(652): }
```

(b) The Newer Version

Fig. 1: Two Versions of a Virtual Devices Written in C. This piece of code is extracted from MAX7310 virtual device in QEMU virtual machine. This code defines a function to reset the registers. However, we do not know why they are resetted to the values labeled 1, 2 and 3. To know the answer, we need to refer to the specification.

3 Our Approach

We first present an overview of our approach. Then, we propose the underlying data model. The data model introduces the requirement of the techniques to represent the POIs and the POSes. We illustrate how our approach address this problem at last.

Table 4. Register 2-Polarity Inversion Register

BIT	I/O7	I/O6	I/O5	I/O4	I/O3	I/O2	I/O1	I/O0
Default	1	1	1	1	0	0	0	0

Table 5. Register 3-Direction Register

BIT	I/O7	I/O6	I/O5	I/O4	I/O3	I/O2	I/O1	I/O0
Default	1	1	1	1	1	1	1	1

Table 6. Register 4-Status Register

BIT	S7	S6	S5	S4	S3	S2	S1	S0
Default	x	x	x	x	x	x	x	1

Fig. 2: Specification Example. This is part of MAX7310 specification. It includes three tables, which define the default values for three registers.

3.1 Overview

this section need rewriting. focus on data.

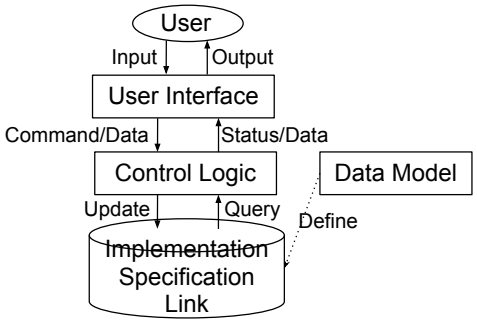


Fig. 3: Framework of Our Approach.

As shown in Fig. 3, the framework of our approach consists three components: database, control logic, and user interface. The structure of the traceability link data in the database is defined by the data model. Control logic lays in the middle of user interface and datamodel. It receives and processes command from the user interface. Depending on the command, it either updates the database, or queries the data in the database and report the result to the user interface. User interface is responsible of communicating with the end user, processs and passes the input from user to the control logic, get the result from the control logic and present the result to the user.

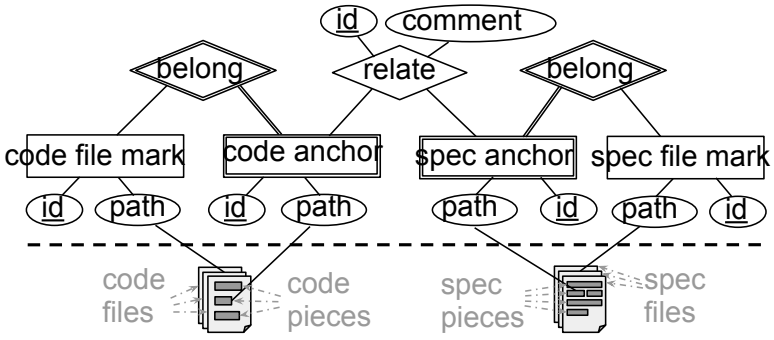


Fig. 4: Data Model. Entity-Relation Model is used. Rectangles represent entity sets. Ovals represent attributes. Diamonds represent relationships.

3.2 Data Model

The data model is the key to support fine-grained links between basic elements of specifications and semantic constructs of implementation languages and automatic migration of links as implementations evolve. We introduce the data model of our approach in this section.

The data model describes the data objects being handled in the framework, their attributes, and the relationships between these data objects. Fig. 4 presents the data model of our design. Entiry-relationship model is used. Rectangles represent entiry sets. Ovals represent attributes. Diamonds represent relationships. We have four entity sets in our design: code file mark, spec file mark, code piece anchor and specification piece anchor. Code piece anchor and specification piece anchor represent the selected code and specification piece respectively. Both code file mark and spec file mark entities have attributes id and path: id is a 128-bit number, and serves as the key of the entity; path is the file path pointing to a file in the local machine. Code piece anchor and spec piece anchor have id and path as attributes: id is a 128-bit number, and serves as the key; path points to the code or spec piece. Traceability links are represented by the relationships between code piece anchor and spec piece anchor.

explain

Special attention should be paid to four attributes: path for code file, path for spec file, path for code piece anchor and path for spec piece anchor. These four attributes are foreign keys used to refer to data that do not exist in database. As foreign keys, these attributes should be able to uniquely identify the entities they refer to. The path provided by the file system is a solution to the path for code files and spec files, as we can easily identify both code files and spec files by these pathes. The pathes for anchors are challenging, little work has been done to address this problem.

3.3 Anchor Design

Anchors includes specification anchor and implementation anchor. The three basic requirements for traceability links are the constraints for the design. We present how our approach make these requirements fulfilled. Specification anchor design is addressed first, then we present the implementation anchor design.

Spec Anchor Specifiations are relatively stable, which make the specification anchor design easier than implementation anchor. Using the absolute coordinate is enough to meet the requirements. Most of the HW/SW interface specification are in PDF format. We use this format to present our approach. Basically, the coordinate for specification piece is denoted by two points – the point representing the begining of the selection and the point representing the end of the selection. Each point is denoted by a vector <page, x-coord, y-coord>, where page is the page number the point located, x-coord and y-coord are the horizontal and vertical coordinate respectively.

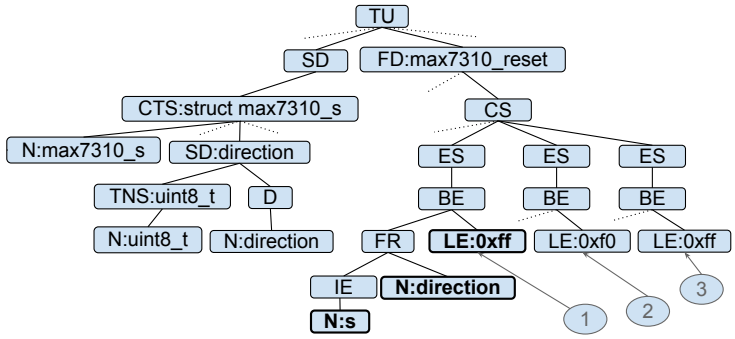
The method is accurate as each point is uniquely represented by the coordinate. This method is also fine-grained, because this method could represent each word, sentence or evan arbitrary size of POS. The method is not robust as it uses the absolute coordinate as the basic element of the anchor. However, specifications are relatively stable, which makes this approach work most of the time.

Offset-Based Anchor Similar to specification anchor, absolute coordinates also fit the situation of implementation anchor. Basically, implementation code is just a string of ASCII charactors. Each piece of code is a substring of this string. We can use the offset of the substring in the string and the length of the piece of code to represent this piece of code. The pair (offset, length) is called Offset-Based Anchor. Fig. 1(a) illustrate this technique. The parenthesized number in front of each line represent the offset of the first charactor of that line. Using this approach, we can get the Offset-Based Anchors for the three code pieces labeled 1, 2 and 3 as (590, 4), (614, 4) and (635, 4) respectively.

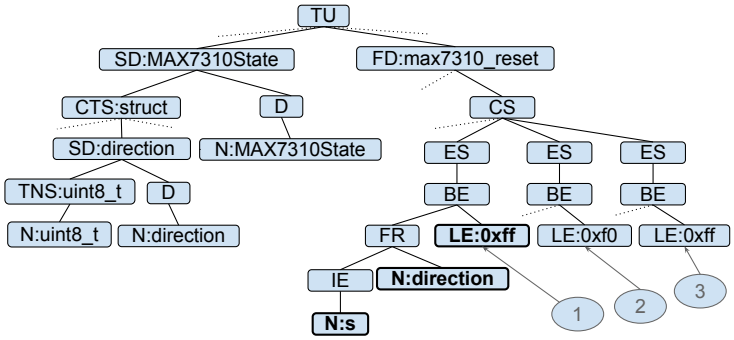
The advantage of this technique is obvious, it is simple and provides fine-grained traceability. The disadvantage of this technique is that these anchors fail after the code evolves. Fig. 1(b) is the code after the code in Fig. 1(a) evolves. We can see the anchors we get above do not point to the previous pieces of code anymore. The reason is that this approach is context sensitive. By saying context sensitive, we mean it depends on its context. In this approach, the anchor depends on the number of charactors preceeding the piece of code.

One way to overcome this disadvantage is to get the knowledge of change to keep the links valid. However, this information is not always available. Most of the time, code is changed by other people. We can only get the changed result without how the change is made.

Syntax Tree Based Anchor To make the traceability links created for one version of code work for other versions, we need a robust strategy to represent the code pieces. More specifically, we need to find some properties of the code that are relatively stable when code evolves. These properties can serve as keys to identify the code piece. We



(a) AST for the Code in Fig. 1(a)



(b) AST for the Code in Fig. 1(b)

```
BE  - BinaryExpression          IE  - IdExpression
CS  - CompoundStatement         LE  - LiteralExpression
CTS - CompositeTypeSpecifier    N   - Name
D   - Declarator                SD  - SimpleDeclaration
ES  - ExpressionStatement       TNS - TypedefNameSpecifier
FD  - FunctionDefinition        TU  - TranslationUnit
FR  - FieldReference
```

Fig. 5: The AST for the implementation code in Fig. 1.

use syntax information to form the main element of our code piece identifier. The code can be parsed and represented by a parse tree or an abstract syntax tree (AST). We use AST in our approach, as it faithfully retains the structure of the original source code, and is concise and efficient to compute compared to parse tree. AST usually has fewer nodes than parse tree, and takes less time to traverse. Each node in AST tree represents a syntax unit of the grammer, a parent of a node represent a bigger grammer unit that includes the grammer unit represented by the child node. The nodes are labeled by the type of the grammer unit. For each leaf node, we also include the value, which is the

code this node represents, in the label. For some of the inner nodes that have a name, we also include the name in their labels. For example, a inner node that represents a function will be labeled with the grammer unit type and the name of the function. A change of the code in one branch of AST usually does not affect the structure in other branches. Representing code pieces based on the structure in the syntax tree is robust. Fig. 5(a) and Fig. 5(b) are the Abstract Syntax Trees (AST) for the code in Fig. 1(a) and Fig. 1(b). Dot lines represent branches not shown. Fig. 5(a) shows two subtrees of the root. The left subtree represents line 13 - line 25 of Fig. 1(a), which defines the structure max7310_s. The right subtree represent line 27 - line 35 of Fig. 1(a), which defines the function max7310_reset. Similarly, the left subtree of the AST in Fig. 5(b) represent line 12 - line 24 in Fig. 1(b), and the right subtree represents line 26 - line 34. From the AST shown in Fig. 5(a) and fig. 5(b), we can see that the left tree has changed as the code for structure definition changed. However, the structure of the right subtree remain the same, unaffected by the change of the left subtree.

Based on the AST of the code, we define our identifier to a piece of code as a sequence of leaf nodes in the AST. Each leaf node is represented by the path from the leaf node to the root of the AST. For example, line 31 in Fig. 1(a) is represented by the highlighted three leaf nodes in Fig. 5(a). The leaf nodes are in the same order as they are visited by depth-first search of the AST. For this example, the closest common parent node for these three leaf nodes also represents the same piece of code, and thes representation is even simpler. We avoid using this representation for two reasons: First, this representation contains less information; Second, this representation is sensitive to formatting and style convention. For example, some programmer may prefer no spaces adjacent to the equal sign, or they may insert some spaces besides the equal sign to make it aligned with other lines of assignment statements. Either case will change the string represented by this common parent, and make the traceability link invalid. We will address how we determine if a traceability link is valid or not in the next section.

Using this approach, let's consider the pieces of code label by 1, 2 and 3 in fig. 1 again. The identifier for these code pieces in AST are shown in fig. 5 labeled with the same number. Through the two ASTs are different, the highlighted nodes can be represented by the same path from the roots of these two ASTs. As a result, we can use this method to identify the piece of code, and the change of the unrelated code does not invalidate the identifier.

This method is relatively stable. However, it may not uniquely identify a piece of code, which means one identifier may refer to two or more pieces of code. The pieces of code label by 1 and 3 in fig. 5(a) showes two pathes that are encoded as the same identifier – <LE:0xff/BE/ES/CS/FD:max7310_reset/TU>. As a result, this identifier can not distinguish these two pieces of code, and can not serve as a key to these pieces of code.

Hybrid Anchor Offset-based anchor can uniquely identify a piece of code, but it is not immune to code evolusion. Syntax tree based anchor is immune to code evolusion, but it can not uniquely identify some pieces of code. We combine these two methods and form hybrid anchor. For each leaf node that has an ambiguous path representation, we can always find at least one of its ancesters that has an unique path. In the worst case, we

will get to the root node. Our approach is composed of two steps. First, we compute the syntax tree based anchor, then, we go over the path from leaf to root, find the first node that is a unique identifier. Second, we compute offset based anchor based on this unique identifier instead of the begining of the whole implementation code. Consider the AST in fig. 5(a), to identify the piece of code labeled by 1, we go through the path towards the root, and get to the first unique identifier at the inner node CS. The node CS represent line 28 - line 35 of the code in fig. 1(a). It is the body of the function `max7310_reset`, and there is only one body for this function by the C language grammer. The offset based identifier based on this node is (103,4), as the first charactor of the node CS is located at offset 487 and the offset based anchor is (590,4). We combine the information of the inner node CS and the calculated coordinate to represent the piece of code with no ambiguity. Same as offset based anchor, if the code does not change, it can uniquely identify the piece of code. If the code changes, this method still works if the code for the unique identifier remain the same. Actually, this is a general approach for the previous two. If we get to the root node while finding the first unique ancestor, then this approach will become the same as offset based anchor. However, if the lead node itself is the firtst node with unique identifier, then this approach is the same as syntax based anchor. The lower the node with the unique identifier in AST, the better our approach works.

4 Implementation

This section presents the architecture of the implementation of our approach first. After that, we introduce the algorithm for validating the traceability links, and for code selection completion which helps the user to complete the code piece selection when part of a syntax element is selected.

4.1 Architecture

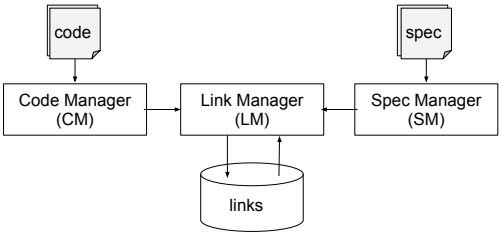


Fig. 6: coDoc Architecture and Features

The architecture of our approach is shown in Figure 6. There are three types of data here, 1) code, 2) spec, and 3) traceability links. These data are consumed by three components: code manager (CM), spec manager (SM) and link manager (LM).

1. *Link Manager.* Traceability link manager get the highlighted code piece from code manager and the highlighted spec piece from spec manager, and create traceability link for these two piece of information. It also reads the traceability link data, verify the validity of the link, and highlights the code and document pieces that are related.
2. *Code Manager.* Code manager reuses C/C++ Development Tool (CDT) editor from Elipse[2] to parses the code, and highlights syntax elements. It can highlight the piece of code based on the information provided by the user or the link manager.
3. *Spec Manager.* Spec manager renders the pdf content, can support select and highlight spec content, and communicate with the link manager.

We implemented coDoc as an Elipse Rich Client Program (RCP), which can make our tool share the appearence in Eclipse that is familiar by many programmer. Figure 7 is the screenshot of the tool. The bottom is the UI for link manager, while the top left part and top right side are the UI for code manager and spec manager repectively.

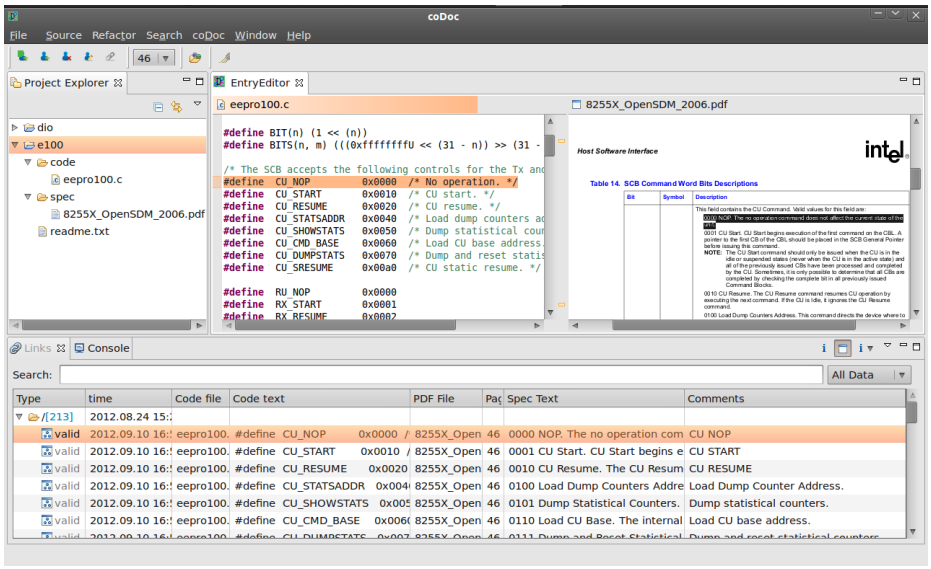


Fig. 7: coDoc Platform View

4.2 Link Validation

When code evolves, we need to validate the existing traceability links. Once an invalid traceability links is found, we should report it to let user correct it manually. To check the validity of the traceability links, we record the content of the code piece and the spec

[2] `http://www.eclipse.org`

Algorithm 1: TRACEABILITY-LINK-VALIDATION($link$)

```
1  AST ← GEN-AST(code)
2  ⟨RES, CONTENT_C⟩ ← GET-CODE-CONTENT(AST, link.code.path)
3  if RES = FAIL then
4      return FAIL
5  RES ← COMPARE-CONTENT(CONTENT_C, link.code.content)
6  if RES = FAIL then
7      return FAIL
8  CONTENT_S ← GET-SPEC-CONTENT(spec, link.spec.path)
9  RES ← COMPARE-CONTENT(CONTENT_S, link.spec.content)
10 if RES = FAIL then
11     return FAIL
12 return SUCCESS
```

piece when generating the links. The content is used later to compare with the content to which the path points to when code changes. The validation process basically go through each traceability link in the database, and check whether the link is valid or not. Algorithm 1 presents the pseudo-code for check the validation of one single traceability link. Line 1 - line 7 check the validity of the code piece part of the traceability link. Line 1 generates the AST for the current code first. Line 2 then invokes the function GET-CODE-CONTENT with the path of the code piece as input, identify the code piece in the new AST, and return the content of the code piece if the code piece is identified successfully. If the path of the code piece cannot identify any code piece or can identify more than one code piece, the function GET-CODE-CONTENT reports failure. Line 5 compare the the new content extracted from the new code with the recorded old content, and fail if the two contents do not match. After finishing check the code piece part, line 8 - line 11 of the algorithm check the spec piece part in a similar flow. If no failure happens, the algorithm return successfully at line 12.

4.3 Code Selection Completion

Algorithm 2: COMPLETE-CODE-SELECTION($selection_{org}$)

```
1  AST ← GEN-AST(code)
2  SEQUENCE_LEAF ← GEN-LEAF-SEQUENCE(AST)
3  POC_AST ← TO-AST-BASED(SEQUENCE_LEAF, selection_org)
4  selection_new ← TO-OFFSET-BASED(POC_AST)
5  return selection_new
```

To improve the user experience of coDoc, we help users to select the code pieces when creating traceability links. Our approach is based on the fact that it is usually meaningless to have part of the basic code elements as the code piece of the traceability links. For example, we do not create a link between half of the variable and the spec piece. We assume that every selected piece of code is composed of valid syntax elements. Based on this assumption, we can help the user to automatically complete their selection when they only select part of the syntax element, like a variable, a constant, and etc. Algorithm 2 outlines our approach. This algorithm takes a code selection in offset based format, and returns a new code selection in offset based format. We choose offset based format because Eclipse use this format to highlight the code piece. The algorithm basically includes two parts: (1) line 1 - line 3 force the input selection to be aligned by transforming it from offset based code piece to AST based code piece. (2) line 4 generates the required format by transforming AST based code piece back to offset based code piece. As we discussed before, AST based representatioin is a sequence of AST leaf nodes in DFS visit order, which is the same order as in the source code. The AST based code piece covers all the meaningful part of the original offset based code piece. To transform the offset based code piece to AST based code piece, we get the sequence of leaf nodes of AST in line 1 and line 2, and then match the leaf node sequence with the code piece to get the AST based representation. The leaf node in the AST based representation is determined this way: if the leaf node contains any part of the selected code, then the leaf node is in the representation. After we get the AST based representation, we transform it back to offset based reprensentation, and return it. This way, each time the user make a selection, our engine adjust the selection to valid syntax boundary.

5 Evaluation

5.1 Experimental Setup

The data in the section are fake.

We evaluate our approach on four virtual device prototypes in QEMU. QEMU is a generic and open source machine emulator, which can host unmodified guest operating systems [5] [4]. QEMU includes virtual implementations for a large number of hardware devices. We do our evaluation on four of them: MAX7310, RTL8139, EEPro100 and E1000. MAX7310 is a 8-bit I/O expander developed by Maxim. This device is used when a system need more I/O ports. EEPro100, E1000 and RTL8139 are three popular network adapters. Both EEPro100 and E1000 are developed by Intel, and RTL8139 is developed by Realtek. All these four devices have specifications in PDF (Portable Document Format) format. We examined the release history of the virtual implementations and specifications of these four hardware device, and found that almost no change has been made to specifications after the corresponding implementations have begun to release.

Based on the QEMU repository, the virtual device prototypes for MAX7310, RTL8139, EEPro100 and E1000 have 22, 143, 163 and 168 versions since they are first created. We check out ten stable versions of all these virtual device drivers from the repository. The ten versions we used are stable-0.10, stable-0.11, stable-0.12, stable-0.13, stable-0.14, stable-0.15, stable-1.0, stable-1.1, stable-1.2 and stable-1.3, we represent them as

1 - 10 repectively. We present the average lines of code (LOC) and the average number of functions of the virtual device prototypes, as well as the number of pages of the specifications in Table 1. The LOC metrics are measured by the tool named locmetrics. The lines of code (LOC) range from 181 to 2771. The sizes of specification range from 15 pages to 410 pages.

The changes from version to version for all the four virtual device prototypes are shown in Table 2. The changes are measured by the lines of code different from version to version using an open source tool cloc.

Table 1: Three Virtual Device Drivers Used for Evaluation

	implementation		specification
	LOC	Functions	pages
MAX7310	181	9	15
RTL8139	2771	101	67
EEPro100	1870	64	175
E1000	1905	52	410

Table 2: The Changes in Ten Versions

	1-2	2-3	3-4	4-5	5-6	6-7	7-8	8-9	9-10
MAX7310	32	28	0	0	18	0	24	0	0
RTL8139	50	349	120	41	432	116	123	9	88
EEPro100	81	765	903	22	304	231	135	21	2
E1000	64	199	39	42	80	120	198	25	27

5.2 Accuracy

We measure the accurary by the number of uniquely identifiable POC over number of traceability links recovered for all the virtual prototypes. The result is shown in Table 3

Table 3: Accuracy

	accuracy
MAX7310	80%
RTL8139	87%
EEPro100	78%
E1000	91%

5.3 Granularity

The identifying method with fine granularity makes the relationship created by coDoc very accurate, 80% of the code pieces are marked inside the statement and expressions. The same result happens for the document. In the whole relationships, 90% of the document pieces are marked on cells in the tables. The granulariy results are listed in Table 4.

Table 4: Granularity Result

	# functions	statements	expressions
MAX7310	108	2	3
RTL8139	213	4	5
EEPro100	213	4	5
E1000	213	4	5

5.4 Adapting to Evolving Code

To evaluate how our approach adapt to evolving code, we recover the traceability links for the first version of all the four virtual device driver code. We then check the number of valid traceability links left in the later versions. The result is shown in table 5.

Table 5: Robustness

	1-2	2-3	3-4	4-5	5-6	6-7	7-8	8-9	9-10
MAX7310	108	2	10	108	2	10	108	2	10
RTL8139	213	4	210	108	2	10	108	2	10
EEPro100	213	4	20	108	2	10	108	2	10
E1000	213	4	290	108	2	10	108	2	10

5.5 Performance

We test the performance of our algorithm to complete code selection and to verify traceability links. The result is shown in table 6.

5.6 Summary

In summary, our approach can meet the three requirements for the traceability links between device drivers and device specifications. We believe this tool will lead to increased productivity.

Table 6: Performance

	verification	completion
MAX7310	108	2
RTL8139	213	4
EEPro100	213	4
E1000	213	4

6 Related Work

improve

Researchers have done much work to build automatic tools for traceability link recovery (TLR) from software systems [2] [7] [13]. TLR technology scans the software artifacts including source code and documentation and recovers the semantic connections by information retrieval (IR) methods. This technology has shown to be advanced and successful [16]. However, IR techniques are not good enough to substitute the human decision-maker in the linking process. Hayes et al. suggests these IR techniques should only be used to generate an appropriate list of candidate links evaluated by software analyst [9]. The recall of these techniques is not as good as to be pratical, a significant part of traceability links need to be recovered manually. Trace acquisition remains human-intensive and with high initial cost as reported in case studies on industrial processes and traceability experiences [12] [15] [3] [8] [14].

The evolution nature of software artifacts makes the situation even worse. No matter how the traceability links are created – automatically or manually, these links are probably only valid for a certain version of the software. As software evolves during the development and maintenance process, software artifacts changes as well. These changes might invalidate the existing traceability links. For automatically recovered links, we can re-run the TLR tool to reestablish them. However, this solution is computationally costly for interactive use during software development stage. For manually recovered links, we must reestablish these links manually again, which is arduous. This intimidate people from using TLR tools in their daily work. Some work has been done to address this problem by improving exisiting IR based methods [10]. However, these work didn't consider the situations specific to hardware related software.

7 Conclusions and Future Work

We present an approach to manage the traceability links between HW/SW interface specifications and implementations, considering the unique properties of these traceability links. We also implemented our approach in a prototype, namely, coDoc, and used this tool to evaluate our approach. We did the experiments on three virtual device prototypes in QEMU. The results show that our approach can effectively manage traceability links for HW/SW interface implementations and their specifications. 70% of the traceability links are fine-grained, 80% of them remains valid after source code evolves to a new version.

In the future, we will continue our research in the following directions. First, we will consider the specification change, and develop strategy to manage traceability links

in this circumstance. Second, we will develop automatic traceability link recovering technique for device drivers and their specifications.

References

1. Abran, A., Nguyenkim, H.: Analysis of maintenance work categories through measurement. In: Software Maintenance, 1991., Proceedings. Conference on. pp. 104–113 (1991)
2. Antoniol, G., Canfora, G., Casazza, G., Lucia, A.D., Merlo, E.: Recovering traceability links between code and documentation. IEEE Trans. Softw. Eng pp. 970–983 (2002)
3. Asuncion, H.U., François, F., Taylor, R.N.: An end-to-end industrial software traceability tool. In: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering. pp. 115–124. ESEC-FSE '07, ACM, New York, NY, USA (2007), http://doi.acm.org/10.1145/1287624.1287642
4. Bellard, F.: Qemu, a fast and portable dynamic translator. In: Proceedings of the annual conference on USENIX Annual Technical Conference. pp. 41–41. ATEC '05, USENIX Association, Berkeley, CA, USA (2005), http://dl.acm.org/citation.cfm?id=1247360.1247401
5. Bellard, F.: Qemu (Jan 2013), http://wiki.qemu.org/Main_Page
6. Chou, A., Yang, J., Chelf, B., Hallem, S., Engler, D.: An empirical study of operating systems errors. In: Proceedings of the eighteenth ACM symposium on Operating systems principles. pp. 73–88. SOSP '01, ACM, New York, NY, USA (2001), http://doi.acm.org/10.1145/502034.502042
7. De Lucia, A., Oliveto, R., Sgueglia, P.: Incremental approach and user feedbacks: a silver bullet for traceability recovery. In: Proceedings of the 22nd IEEE International Conference on Software Maintenance. pp. 299–309. ICSM '06, IEEE Computer Society, Washington, DC, USA (2006), http://dx.doi.org/10.1109/ICSM.2006.32
8. Gotel, O., Finkelsteiin, A.: Extended requirements traceability: Results of an industrial case study. In: Proceedings of the 3rd IEEE International Symposium on Requirements Engineering. pp. 169–. RE '97, IEEE Computer Society, Washington, DC, USA (1997), http://dl.acm.org/citation.cfm?id=827255.827842
9. Hayes, J.H., Dekhtyar, A., Sundaram, S.K.: Advancing candidate link generation for requirements tracing: The study of methods. IEEE Trans. Softw. Eng. 32(1), 4–19 (Jan 2006), http://dx.doi.org/10.1109/TSE.2006.3
10. Jiang, H.Y., Nguyen, T.N., Chen, I.X., Jaygarl, H., Chang, C.K.: Incremental latent semantic indexing for automatic traceability link evolution management. In: Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering. pp. 59–68. ASE '08, IEEE Computer Society, Washington, DC, USA (2008), http://dx.doi.org/10.1109/ASE.2008.16
11. Lawrence, P.S., Pfleeger, S.L., Atlee, J.M.: Software engineering: theory and practice. Pearson Education India (2006)
12. Lindvall, M., Sandahl, K.: Practical implications of traceability. Softw. Pract. Exper. 26(10), 1161–1180 (Oct 1996), http://dx.doi.org/10.1002/(SICI)1097-024X(199610)26:10<1161::AID-SPE58>3.3.CO;2-O
13. Marcus, A., Maletic, J.I.: Recovering documentation-to-source-code traceability links using latent semantic indexing. In: Proceedings of the 25th International Conference on Software Engineering. pp. 125–135. ICSE '03, IEEE Computer Society, Washington, DC, USA (2003), http://dl.acm.org/citation.cfm?id=776816.776832

14. Neumuller, C., Grunbacher, P.: Automating software traceability in very small companies: A case study and lessons learne. In: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering. pp. 145–156. ASE '06, IEEE Computer Society, Washington, DC, USA (2006), http://dx.doi.org/10.1109/ASE.2006.25
15. Ramesh, B., Powers, T., Stubbs, C., Edwards, M.: Implementing requirements traceability: a case study. In: Proceedings of the Second IEEE International Symposium on Requirements Engineering. pp. 89–. RE '95, IEEE Computer Society, Washington, DC, USA (1995), http://dl.acm.org/citation.cfm?id=827254.827798
16. Spanoudakis, G., Zisman, A.: Software traceability: A roadmap. In: Handbook of Software Engineering and Knowledge Engineering. pp. 395–428. World Scientific Publishing (2004)
17. Swift, M.M., Bershad, B.N., Levy, H.M.: Improving the reliability of commodity operating systems. In: Proceedings of the nineteenth ACM symposium on Operating systems principles. pp. 207–222. SOSP '03, ACM, New York, NY, USA (2003), http://doi.acm.org/10.1145/945445.945466