Proceedings of the Federated Conference on Computer Science and Information Systems pp. 1727–1736
DOI: 10.15439/2016F173
ACSIS, Vol. 8. ISSN 2300-5963

Managing Big Clones to Ease Evolution: Linux Kernel Example

Kuldeep Kumar
Department of Computer Science and Information Systems,
BITS-Pilani, Pilani Campus, India
kuldeep.kumar@pilani.bits-pilani.ac.in

Stan Jarzabek
Faculty of Computer Science
Bialystok University of Technology,
Poland
s.jarzabek@pb.edu.pl

Daniel Dan
Info-Software Systems
ST Electronics Pte. Ltd.,
Singapore
ddan8807@gmail.com

***Abstract*—Successful software is often enhanced and adapted to the needs of new users. During evolution, a software system grows in size, becomes more complex, and costly to maintain. In this paper, we point to big clones—large granular duplicated program structures such as files or directories—as one of many reasons why this happens. Using the Linux kernel as an example, we show that big clones arise in the Linux kernel despite careful architecture design and a systematic approach for managing variability. We propose a solution to avoid these big clones by representing them as generalized templates in ART (Adaptive Reuse Technique). ART templates are constructed on top of the Linux code, without conflicts with the state-of-art techniques and tools used to manage the Linux kernel. Benefits include simplification of the Linux kernel due to non-redundancy, easier comprehension, and traceability of the change impact during evolution. The proposed technique is general and the Linux example discussed in this paper also illustrates general phenomena.**

I. Introduction

CHANGES in user features, design decisions, and platforms arise naturally during software evolution. Sometimes, these changes are contained at the architectural level [1]. But, most often the impact of changes spreads widely throughout the code. It has been reported that after years of evolution software systems grow in size, their structure decays, and become more and more difficult to maintain [2]. Evolution may lead to cloning [3]. New system versions are generally built by cloning (copy-paste-modify practice) code from the earlier versions. However, less cloning happens in advanced Software Product Line (SPL) solutions [4] where reuse and evolution are aided by systematic variability management rather than by cloning. There is a large body of research on reasons why clones arise—both within and across system versions—and whether clones are good or bad [5][6][7]. These studies show that designers may intentionally create certain clones to fulfill some design goals (e.g., for performance, readability, or yet other reasons) [5]. Other clones may result from careless design and can be refactored [6][8], and yet others may not play any useful role, but cannot be eliminated using conventional design techniques [7]. Nevertheless, cloning is a reality and there is need to deal with it [9]. No matter if clones are good or bad, it is beneficial to know where clones are in programs. It is particularly true for big clones such as duplicated files or directories. Big clones happen even if software evolution is systematically managed with variability management techniques [10]. In the paper, we use the Linux kernel to illustrate why this happens, and how we can manage big clones.

The Linux kernel is among the largest well documented evolving systems systematically managed with variability techniques [11]. In that sense, a family of the Linux-kernel versions forms an SPL whose reusable core assets include a carefully designed architecture, systematically identified and documented configuration options (SPL features), a code base managed with the C preprocessor (cpp) as a main variability technique, Kconfig, and other tools and techniques. The reason why we find big clones in the Linux kernel—and, we believe, in many other evolving systems—is that commonly used variability management techniques fail to avoid them in a convenient way.

Using generics (or C++ templates), we can non-redundantly represent similar classes differing in type parameters [12]. Big clones found in industrial systems need not be classes, but files or directories–program structures of any kind and size that differ in arbitrary ways, not just in type parameters. In this paper, we use Adaptive Reuse Technique (ART: https://sourceforge.net/projects/vclang/) to represent big clones as generalized templates. Like generics or C++ templates, ART templates can be instantiated in variant forms. Unlike other templates, ART templates can be built for groups of similar program structures of any kind (e.g., files or directories) that differ in variety of ways typically found in real systems. ART is an enhanced, lightweight and XML-free version of XVCL [13].

We briefly introduce variability management in the Linux kernel in Section II. In Section III, we discuss examples of big clones in Linux kernel version 3.10. Section IV describes how ART blends into Linux kernel development and use cycles. Sections V and VI describe explain how we manage big clones with ART. We evaluate the benefits and trade-offs of the proposed solution in Section VII. Related work and conclusions end the paper.

II. Variability Management in the Linux Kernel

Despite technological advancements in programming technologies, preprocessors are still indispensable. Preprocessing solves some niche problems better than other techniques do. One such problem area is variability management in software evolution and reuse. To some extent

we can manage variability at the level of software architecture [1]. But from architecture variability leaks to the code, and here that preprocessors along with configuration files and other similar techniques [14] become handy.

The Linux kernel developers applied clean architectural design, cpp, build tools, shell scripts, and other tools to facilitate adaptation of the Linux kernel to the specifics of target computers. Close to eleven thousand configuration options control the adaptation process. These options correspond to cpp parameters that navigate execution of cpp directives (such as #ifdef's) that select code relevant to the target computer of user's choice. The high-level configuration tool Kconfig maps these configuration options to the chains of relevant cpp directives embedded in the Linux code. Having selected required options, Kconfig automatically triggers execution of cpp directives and selects compilation units via the make utility to build a custom Linux kernel for a specific computer. Users do not have to understand the details of the adaptation mechanism. While the complexities of cpp instrumentation are hidden from the users, developers who maintain and extend the Linux kernel must understand code instrumented with cpp.

III. Motivating Example: Big Clones in the Linux

In the Linux kernel, the Journaling Block Device (JBD) provides an interface for the file system journaling. There are two directories namely /jbd and /jbd2 implementing this functionality, with /jbd2 being an evolutionary branch of /jbd. /jbd2 compatibly extends /jbd with new features such as support for 64-bit computers, check-summing of journal transactions, and asynchronous transaction commit block write.

Each directory consists of six files shown in Fig. 1. Much similarity in functionality and code (Table I) among files corresponding by names suggests that /jbd2 files were created by copying and modifying /jbd files. Fig. 2 sketches code snippets highlighting the code similarities and differences between the two checkpoint.c files.

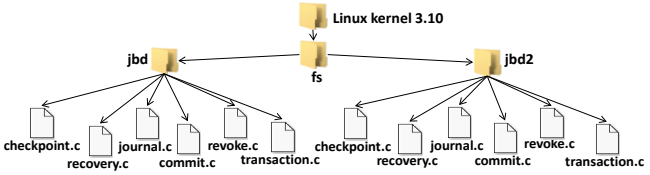

Fig. 1. Motivating example: cloned directories /jbd and /jbd2

Identical Code Fragments : ~554 LOC

```
51: static inline void __buffer_unlink(struct journal_head *jh)
52: {
53:         transaction_t *transaction = jh->b_cp_transaction;
54:
55:         __buffer_unlink_first(jh);
56:         if (transaction->t_checkpoint_io_list == jh) {
57:                 transaction->t_checkpoint_io_list = jh->b_cpnext;
58:                 if (transaction->t_checkpoint_io_list == jh)
59:                         transaction->t_checkpoint_io_list = NULL;
60:         }
61: }
```

```
51: static inline void __buffer_unlink(struct journal_head *jh)
52: {
53:         transaction_t *transaction = jh->b_cp_transaction;
54:
55:         __buffer_unlink_first(jh);
56:         if (transaction->t_checkpoint_io_list == jh) {
57:                 transaction->t_checkpoint_io_list = jh->b_cpnext;
58:                 if (transaction->t_checkpoint_io_list == jh)
59:                         transaction->t_checkpoint_io_list = NULL;
60:         }
61: }
```

Code Fragments with Parametric Changes: ~47 LOC

```
128: while (__log_space_left(journal) < nblocks) {
129:         if (journal->j_flags & JFS_ABORT)
130:                 return;
131:         spin_unlock(&journal->j_state_lock);
132:         mutex_lock(&journal->j_checkpoint_mutex);
```

```
124: while (__jbd2_log_space_left(journal) < nblocks) {
125:         if (journal->j_flags & JBD2_ABORT)
126:                 return;
127:         write_unlock(&journal->j_state_lock);
128:         mutex_lock(&journal->j_checkpoint_mutex);
```

Code Modification: ~12 LOC

```
333: set_buffer_jwrite(bh);
334: bhs[*batch_count] = bh;
335: __buffer_relink_io(jh);
336: jbd_unlock_bh_state(bh);
337: (*batch_count)++;
338: if (*batch_count == NR_BATCH) {
339:         spin_unlock(&journal->j_list_lock);
340:         __flush_batch(journal, bhs, batch_count);
```

```
311: journal->j_chkpt_bhs[*batch_count] = bh;

312: __buffer_relink_io(jh);
313: transaction->t_chp_stats.cs_written++;
314: (*batch_count)++;
315: if (*batch_count == JBD2_NR_BATCH) {
316:         spin_unlock(&journal->j_list_lock);
317:         __flush_batch(journal, batch_count);
```

Code Insertion: ~29 LOC

```
306: spin_unlock(&journal->j_list_lock);
```

```
276: transaction->t_chp_stats.cs_forced_to_close++;
277: spin_unlock(&journal->j_list_lock);
278: if (unlikely(journal->j_flags & JBD2_UNMOUNT))
279:         /* The journal thread is dead; so starting and
281:          * waiting for a commit to finish will cause
282:          * us to wait for a _very_ long time.*/
284:         printk(KERN_ERR "JBD2: %s: "
285:                 "Waiting for Godot: block %llu\n",
286:                 journal->j_devname,
287:                 (unsigned long long) bh->b_blocknr);
```

Code Deletion: ~95 LOC

```
520: journal_update_sb_log_tail(journal, first_tid, blocknr,
521:         WRITE_FLUSH_FUA);
522: spin_lock(&journal->j_state_lock);
523: /* OK, update the superblock to recover the freed space.
524:  * Physical blocks come first: have we wrapped beyond the end of
525:  * the log? */
526: freed = blocknr - journal->j_tail;
```

```
460: __jbd2_update_log_tail(journal, first_tid, blocknr);
```

Fig. 2. Motivating example: code snippets of cloned file /jbd/checkpoint.c (left) and /jbd2/checkpoint.c (right)

TABLE I. SIMILARITY AMONG FILES IN DIRECTORIES /JBD2 AND /JBD

File Name	Total LOC in corres-ponding jbd/jbd2 files	Identical LOC	LOC with parametric differences	Modified LOC	Inserted LOC	Deleted LOC
checkpoint.c	782/705	554	47	12	29	95
commit.c	1002/1192	523	93	35	364	218
journal.c	2122/2146	1266	287	29	690	229
recovery.c	594/862	420	52	12	234	0
revoke.c	740/769	544	94	3	25	0
transaction.c	2229/2348	1346	130	56	516	399

The directories /jbd and /jbd2 exemplify the situations that can benefit from ART as they cannot be effectively handled by other techniques. The reasons why we find such situations in the Linux kernel are functional similarities among different subsystems, extensions to the existing functionalities, adaptation of the existing subsystem code for the new one (incremental development), evolutionary development, and decentralized development [15][16][17].

IV. ART FOR THE LINUX KERNEL

In this section, we show how ART blends with the Linux-kernel development and uses cycles (Fig. 3). A *Linux Developer*, a member of an open-source community evolves the Linux kernel, e.g., by adding new devices into it. The *Linux SysAdmin* adapts the kernel for her computer using tools such as Kconfig.

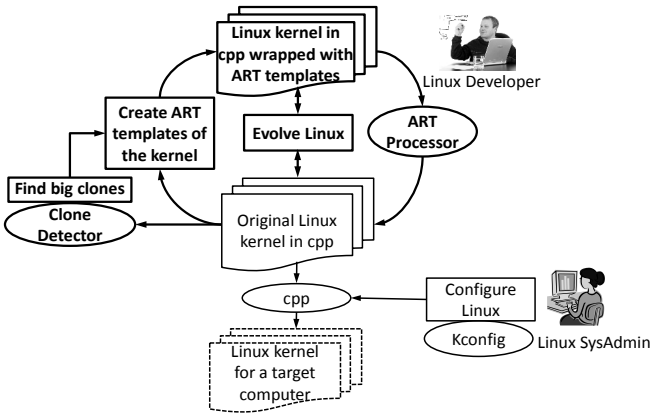

Fig. 3. An overview of Adaptive Reuse Technique (ART)

Big clones can be identified in the Linux kernel with aid of a suitable clone detector (we used Clone Miner [18]). The Linux Developer builds ART templates for big clones on top of the Linux code managed by cpp. From that point onwards, big clones are maintained via ART templates. ART templates do not affect the work of the Linux SysAdmin.

The ART Processor converts ART templates back to the original Linux code. The ART Processor instantiates templates in the same way as the C Preprocessor expand cpp directives. For example, for a template representing a group of similar files, the ART Processor generates code for these files based on specifications of deltas—differences between the template and each of these files. The generated Linux code is in the original form, and can be processed as normal by Kconfig, cpp, or make tool.

ART-template view of the Linux kernel and the original Linux kernel can be used together in two independent cycles of maintaining and using the kernel.

V. TYPES OF CLONES THAT WE HANDLE WITH ART

We categorized big clones in the Linux kernel based on their granularity.

A. Similar Directories

In the example of Section III, /jbd and /jbd2 play the same role, with /jbd2 being an evolutionary branch of /jbd addressing a new computer architecture and its capabilities. Each of the two directories contains six files, with much similarity between files corresponding by names. We found five other cases in Linux kernel following the pattern of /jbd and /jbd2, with the number of files in these directories varying between 19 and 46. In some cases, a directory contained one or more files that do not have similar counterparts in the cloned directory.

B. Similar Files

We found many cases of similar files within the same directory, as well as across directories. A common reason for replicating a file in the same directory is to make a certain existing functionality available for yet other computer architecture, device, or tool. An example is drivers for different brands of touchscreen devices—in directory /drivers/input/touchscreen, 10 files share the same structure and much code. Two directories having almost similar purpose (vide our motivating example) may contain similar files. Sometimes, the same or similar file may be required in two or more directories, even if these directories have not enough code similarity. For example, functionality for handling extended user attributes is needed in directories /fs/ext2, /fs/ext3 and /fs/ext4, therefore file "xattr_user.c" that defines this functionality appears in all three directories.

C. Duplicated Code Fragments

At times, creating templates for duplicated code fragments can be useful too, provided these fragments are long enough, play some specific role (e.g., represent some meaningful function), or recur in many places in programs. For example, code fragments in Fig. 4 implement a device specific queue handling procedure for different wireless network adapters. An instance of this code fragment occurs once in each of the files "rt2400pci.c", "rt2500pci.c", "rc2800pci.c" and "rt61pci.c", and twice in each of the files "rt2500usb.c", "rc2800usb.c" and "rt73usb.c".

VI. CONSTRUCTION AND PROCESSING OF ART TEMPLATES

In this section, we explain how we represented big clones as ART templates. We start with a brief overview of how ART works, followed by the explanation of how we build ART templates, illustrated with the Linux kernel example.

rt73usb.c

```
static void rt73usb_start_queue(struct data_queue *queue) {
        struct rt2x00_dev *rt2x00dev = queue->rt2x00dev;
        u32 reg;
        switch (queue->qid) {
        case QID_RX:
                rt2x00usb_register_read(rt2x00dev, TXRX_CSR0, &reg);
                rt2x00_set_field32(&reg, TXRX_CSR0_DISABLE_RX, 0);
                rt2x00usb_register_write(rt2x00dev, TXRX_CSR0, reg);
                break;
        case QID_BEACON:
                rt2x00usb_register_read(rt2x00dev, TXRX_CSR9, &reg);
                rt2x00_set_field32(&reg, TXRX_CSR9_TSF_TICKING, 1);
                rt2x00_set_field32(&reg, TXRX_CSR9_TBTT_ENABLE, 1);
                rt2x00_set_field32(&reg, TXRX_CSR9_BEACON_GEN, 1);
                rt2x00usb_register_write(rt2x00dev, TXRX_CSR9, reg);
                break;
        default:
                break;
        }
}
```

rc2800usb.c

```
static void rt2800usb_start_queue(struct data_queue *queue) {
        struct rt2x00_dev *rt2x00dev = queue->rt2x00dev;
        u32 reg;
        switch (queue->qid) {
        case QID_RX:
                rt2x00usb_register_read(rt2x00dev, MAC_SYS_CTRL, &reg);
                rt2x00_set_field32(&reg, MAC_SYS_CTRL_ENABLE_RX, 1);
                rt2x00usb_register_write(rt2x00dev, MAC_SYS_CTRL, reg);
                break;
        case QID_BEACON:
                rt2x00usb_register_read(rt2x00dev, BCN_TIME_CFG, &reg);
                rt2x00_set_field32(&reg, BCN_TIME_CFG_TSF_TICKING, 1);
                rt2x00_set_field32(&reg, BCN_TIME_CFG_TBTT_ENABLE, 1);
                rt2x00_set_field32(&reg, BCN_TIME_CFG_BEACON_GEN, 1);
                rt2x00usb_register_write(rt2x00dev, BCN_TIME_CFG, reg);
                break;
        default:
                break;
        }
}
```

Fig. 4. Sample code fragments from rt73usb.c and rc2800usb.c (differences highlighted)

A. An Overview of ART

For each group of clones, we distill common code into ART templates and mark the locations where clones differ one from another with ART commands (*italicized* for clarity in the description below). Fig. 5 outlines the overall solution, which consists of an ART-template hierarchy in which templates at the lower-level serve as building blocks for the higher-level templates. The ART templates are linked together by *#adapt* commands. The top-most template, called the specification file (SPC), specifies how to adapt other templates lower in the hierarchy to accommodate required variations. The ART Processor checks the templates for their conformance to the ART grammar definitions. It then traverses the template hierarchy in the depth-first order, starting with the SPC and performs adaptations by executing the ART commands embedded in the SPC and other ART templates. During traversal, each ART template adapts other templates from its sub-hierarchy. At the end, the ART Processor produces the required cloned instances.

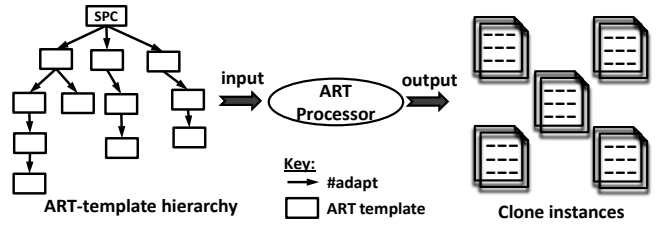

Fig. 5. An overview of the ART-template solution

Fig. 6 depicts steps in template processing. The ART Processor starts by reading the SPC (step-1). It fetches the ART commands step-by-step in the order in which they appear in the SPC (step-2). Whenever it hits an *#adapt* command (step-3), the processing will switch immediately to the adapted template (step-4) and switch back when the adapted template finishes its processing. Within a template, each ART command is processed one after another, in the same way as in the SPC. For the other commands, the Processor executes the ART command and builds the output (step-4') incrementally. Once the Processor reaches the end of the SPC (step-5), it generates the required source code files (step-6); if not, the ART Processor fetches the next ART command from the SPC (step-6').

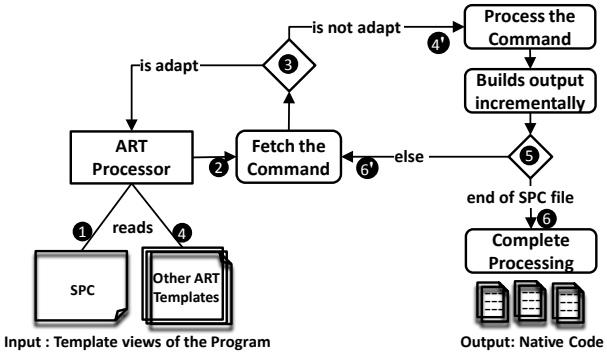

Fig. 6. Traversal mechanism of the ART Processor

B. ART-template Construction Mechanism

Despite a large fraction of the code common to all the clone instances (i.e., identical code fragments in the corresponding clone instances), as shown in Fig. 2, the three main types of differences among corresponding clone instances are parametric differences (code with parametric changes), alternatives (code modifications), and extras (code insertions and deletions).

The first task during the ART-template construction process is to identify these similarities and differences among corresponding clone instances. Once the corresponding similarities and differences are identified, ART templates record exact locations of these variation points at which the clone instances differ. ART commands can be used systematically to mark these variation points. Identical code fragments can be used directly as-it-is in the corresponding ART templates. ART variables treat parametric differences. The ART command *#select* allows choosing one among pre-defined alternatives (options), and *#insert* into *#break* mechanism handles additions and deletions of extra code.

C. Example: Template Construction for JBD

Fig. 7 shows the structure of ART solution for the JBD files. Each pair of similar files (e.g., checkpoint.c in /jbd and /jbd2) is represented by a template (e.g., checkpoint.art). The associated template checkpoint.spc specifies the differences between the two source files as deltas from checkpoint.art. The top-most template jbdX.spc navigates the process of instantiating the templates to form the Linux source files in their original form (i.e., instrumented with cpp).

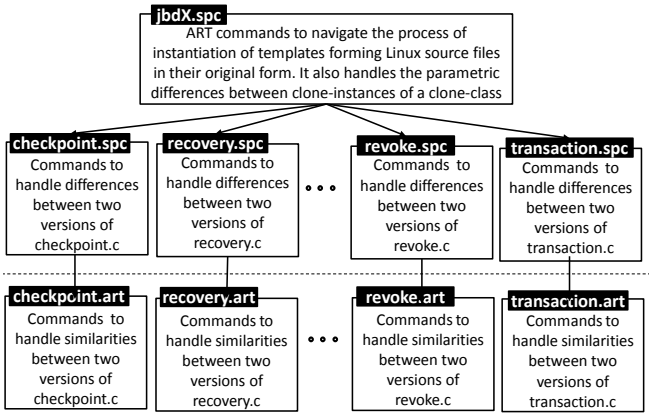

Fig. 7. Constructing ART templates: JBD example

Fig. 8 shows the details of ART templates. In jbdX.spc, ART variables are declared using *#set* commands (lines 1–6). Variable "dirName" is assigned two values, "jbd" and "jbd2" (line 2) that control the *#while* loop (line 7). The loop executes twice, with the value of "dirName = jbd" in the first iteration, and the value of "dirName = jbd2" in the second one. The Variable "fileName" is set to six values, each representing a file name (line 3).

The ART variable "action" helps represent lines:

*spin*_unlock(&journal->j_state_lock); //in jbd/checkpoint.c

*write*_unlock(&journal->j_state_lock); //in jbd2/checkpoint.c

in a single line in checkpoint.art (line 4):

*?@action?*_unlock(&journal->j_state_lock);

The two values of "action" are defined by:

#set action = "spin", "write" // line 4 in jbdX.spc

The generation loop defined in line 7:

#while dirName, action,…, tagByte

is controlled by a list of variables that cater for all parametric differences between the two checkpoint.c files. The command *#output* (line 9) instructs the ART Processor to create a directory and to place any further output into this directory (if the output file or directory is not specified, the ART Processor emits the code to an automatically generated default file named "defaultOutput"). Expression "?@fileName?" is used to fetch the value of an ART variable filename (line 9).

Similar to cpp's #include directive, an *#adapt* command (line 10 in jbdX.spc) instructs the ART Processor to include the designated template to the output. In addition, the *#adapt* command also tells the Processor to customize the designated template and assemble the customized result into the output. For example, given two ART templates *t* and *t'*, the statement "*#adapt t*" in template *t'* suspends processing of the current template (i.e., *t'*), and transfers processing to the template *t*. The ART Processor applies all the customizations specified under template *t*. Commands below *#adapt* in template *t'* indicate customizations to be applied after the template *t* is processed.

Variation points at which the two corresponding files (e.g., checkpoint.c) in /jbd and in /jbd2 directories differ are marked using ART commands—references to the ART variables, *#select*, *#break*, and possibly other commands. ART variables control selection of the code in case of alternative differences. This is illustrated as "*#select* dirName" in the template checkpoint.spc (line 4). *#option* (line 5 and 10 in checkpoint.spc) controls the variable values.

File checkpoint.c in one directory contains some extra lines as compared to checkpoint.c in another directory. These extra lines are specified using *#insert* commands in various "*#select* dirName" options. "*#insert* process_buffer" (line 11 in checkpoint.spc) propagates the code to "*#break* process_buffer" in checkpoint.art (line 12). *#insert-before* and *#insert-after* (line 6–9 in checkpoint.spc) add their code before or after the code contained in the matching *#break* (line 7 in checkpoint.art). While *#select* instruments a template with known variations, *#break* allows for extensions to a template in unexpected ways in the specific context of adaptation, without affecting others. These provisions for unexpected evolutionary changes give ART templates flexibility and stability.

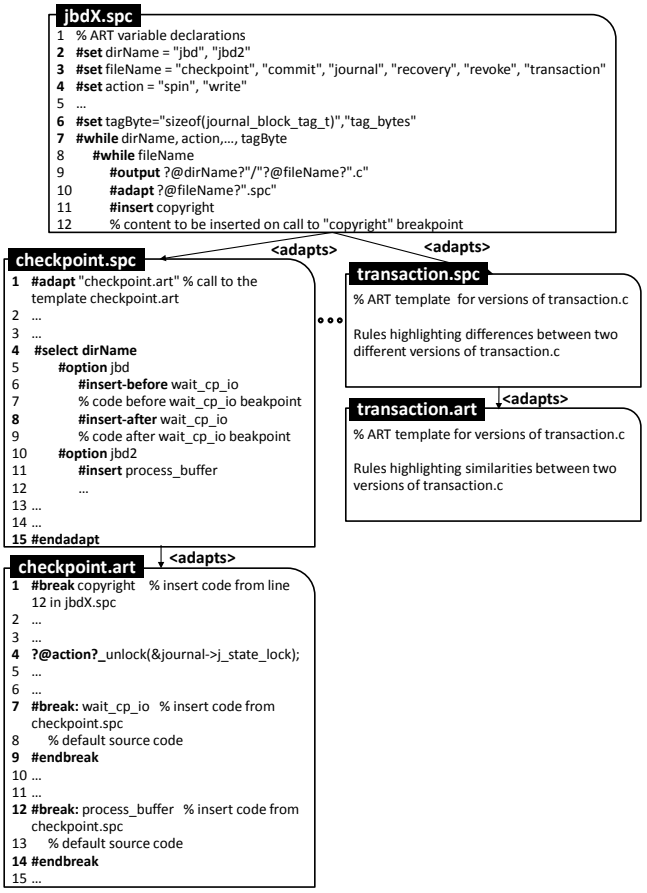

Fig. 8. Code snippet of ART templates for the JBD example

The ART Processor generates Linux code traversing the template hierarchy and emitting the code for the six files in the /jbd and /jbd2 directories from their respective templates. After that, the Linux code can be configured with Kconfig, and processed with cpp in the usual way. Template views expose the fact that the two directories and corresponding files in them are similar to each other, and also explicate every detail of similarities and differences among them. This information is implicit in the Linux code. Explicating it using

ART can be useful in further evolution of the JBD file system (Section VII).

D. Other Similarity Patterns at the Directory Level

Other cases of similar directories may not follow such a regular similarity pattern as in /jbd and /jbd2. For example, in the directories /drivers/infiniband/hw/qib and /drivers/infiniband/hw/ipath, in addition to similar files, /drivers/infiniband/hw/qib contains some extra files that do not have a counterpart in /drivers/infiniband/hw/ipath. Still, there is enough similarity in the concept and code between /drivers/infiniband/hw/ipath and /drivers/infiniband/hw/qib to build an ART-template solution for these two directories. The scheme used for building ART templates for /jbd and /jbd2 is also applicable in these situations, as templates manage pairs of similar files only and the remaining other files remain intact in the directories.

E. Constructing Templates for Similar Files

In this case, we deal with the similar files found in the same directory and the similar files in different directories, bearing in mind that directories as a whole are not considered good candidates for representing them as templates. For each such situation, we can create ART templates for similar files if we think that exposition of similarities and differences among these files can aid developers in reuse, program understanding, maintenance, and evolution of the Linux kernel. The solution follows the similar scheme as shown in Fig. 7 and Fig. 8.

VII. Evaluation

ART and its predecessor XVCL have been applied in industrial projects as a variability technique to manage reuse in product lines of web portals, and command and control systems [19][20]. In these projects, the productivity impact of applying the technique was measured and evaluated. An industry partner also participated in the Linux study described in this paper. In this section, we evaluate our ART solution for Linux, complementing it with lessons learned from other industrial projects with ART.

A. Reusing Templates within a Version of the Linux kernel

In a large system such as the Linux kernel, it is common to find clones within subsystems or modules, as well as across subsystems or modules. Each clone group can be managed by ART templates as long as such a non-redundant representation is deemed useful. Therefore, ART solution takes form of template hierarchies (Fig. 9) that explicates the location of clones and the exact nature of similarities and differences among replicated program structures. This knowledge is generally useful in understanding program design.

The example in Fig. 9 shows how ART templates reveal implicit couplings among bigger structures that contain repetitions. The same functionality defined in the templates commonConnectDisconnect.art and serioDriverStructure.art is needed in /touchscreen/common.art and /joystick/common.art. Templates for these two directories explicitly show the fact that this functionality is needed in both "touchscreen" and "joystick" drivers. If such implicit dependency among program modules is not documented, it may be overlooked during program evolution that may lead to errors.

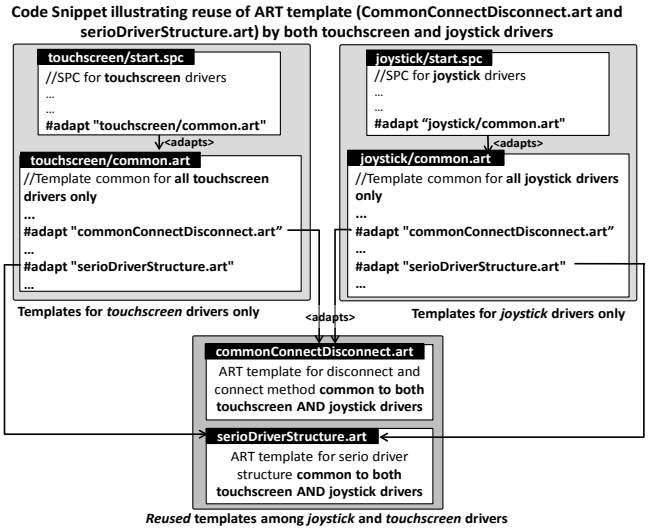

Fig. 9. Template reuse

B. Reusing Templates across Versions of the Linux kernel

Template reuse interconnects ART-template solutions developed for different groups of clones from the bottom, as shown in Fig. 9. It is also useful to interconnect partial ART-template solutions from the top, by introducing higher-level umbrella templates that trigger ART processing of some or all templates in the solutions.

Umbrella templates help developers manage multiple versions of the Linux kernel from the common base. A case study performed on 136 stable versions of the Linux kernel shows clone coverage of approximately 67% [21]. The coverage was found to be even higher between two consecutive versions due to small changes in successive releases of the kernel. Using umbrella templates, as shown in Fig. 10, we represented the commonalities between two versions, together with the version-specific code in different templates.

C. Handling Evolutionary Changes

Evolution often brings forward changes to the requirements and related code. For example, there might be a need to add a new directory /jbd3, or add more files to the JBD directories. ART has provisions to accommodate evolutionary changes to the templates (e.g., adding jbd3), without affecting existing code derived from the templates (e.g., jbd and jbd2).

Assuming that the new directory /jbd3 also contains six files that are similar to their counterparts in the /jbd and /jbd2, we need to make the following changes to the templates shown in Fig. 8:

jbdX.spc:

#set dirName = "jbd", "jbd2", **"jbd3"**

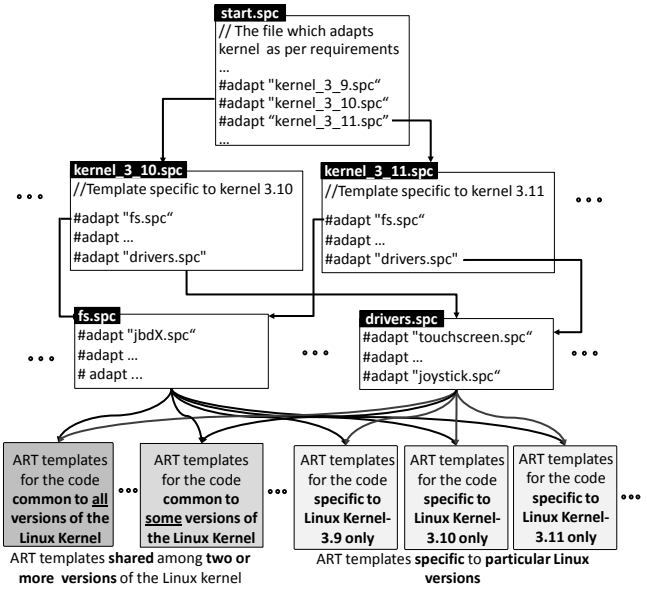

Fig. 10. Umbrella templates for an overall ART solution

```
#set fileName = "checkpoint",…, "recovery"
…
#while dirName , action,…, tagByte
#while filename
#output ?@dirName?"/"?@fileName?".c"
   #adapt: ?@fileName?".spc"
   …
```

checkpoint.spc:

```
#adapt: "checkpoint.art"
#select dirName
#option jbd3
…
% Customization to other templates with regards to jbd3
```

checkpoint.art:

```
% Customizations to checkpoint.art specific to jbd3
% Customizations to other templates considering jbd3
```

In case of new variation points between the template and the file in /jbd3, we place new *#break* commands in the template. These new *#break* commands will cater for the differences specific to /jbd3, injected by *#insert*s in "*#option* jbd3" without affecting /jbd or /jbd2.

D. Aid in Program Understanding and Maintenance

Ease of comprehending program relations that matter during maintenance: ART templates enhance important relationships among program elements that matter to programmers trying to understand and modify the code. Instead of dealing with each directory or file separately, programmers can comprehend them in groups, and see the commonalities and differences among members of each group. It reduces ripple effects and the risk of update anomalies. In this way, if one wants to change a file, it is easy to check whether the changes also affect other files. For example, as illustrated in Fig. 7, similarities and differences are explicitly visible for jbd and jbd2 file systems. Such relations are generally hidden in conventional programs. Making them visible and easily tractable improves program maintenance. It also makes impact of changes easy to comprehend (as shown in Section VII.C).

Non-redundancy: ART templates eliminate redundant code from the software systems. For example, in our Linux experiment, ART templates reduced the size of code with redundancies by 30-50%. As both code and comments are important in software maintenance and program understanding, the upside of the proposed technique is that it is possible to manage both duplicated code and comments using it. ART allows a clean separation of various sources of changes that affect program during evolution. ART templates reduce the number of points at which affected changes must be made. Changes done to one template consistently propagate to all the contexts in which that template is adapted. Even if the changes are not uniform, adaptations can be made at specific variation points using the ART commands without directly modifying the code fragments. The ART template hierarchy explicitly reflects the impact of changes on the program structure. We can easily trace how different features affect the code.

Enhancing program understanding and conceptual integrity: According to Brooks [22], program understanding and conceptual integrity are among the most important considerations in system design. Big clones often embody domain-specific abstractions or design concepts. By formally capturing these abstractions and concepts, ART templates aid in program understanding and enhance conceptual integrity of the design.

Creating templates can be considered as refactoring at the meta-level: In some cases, developers seek to improve certain program qualities but due to some unavoidable reasons cannot achieve this at the code level. In such cases, we can do that at the level of meta-level templates. We benefit from non-redundancy at the meta-level templates, while still keeping repetitions in programs (as it is often desirable or unavoidable [7][23]).

Formally representing multiple design views: Program modules often belong to many logical groups that matter to developers at different times. Each logical partitioning reflects a certain aspect of program design that matters at a given time in the development in a given context. For example, for a given business function in business software, the modules for user interface, business logic, and database are usually implemented in different system partitions. Logically these modules belong to each other and sometimes we must know which modules implement a given business function completely. But, only one logical partitioning can be formally represented in a program physical structure. ART provides means to overlay programs with a web of meta-structures formally defining these logical partitions linked to code and without conflicts with the code.

Other Benefits: ART makes it easy for the programmers to do any program modifications and extensions at the template level. So there is no need to modify the code, and templates always remain in sync with the code as programs evolve.

In addition, with reference to the Linux kernel, aiding Linux Developers without affecting Linux SysAdmins is one cornerstone of the proposed solution. ART templates are non-intrusive, i.e., they do not affect the way cpp is normally used, but it improves understandability and maintainability of the programs instrumented with cpp. Also, the Linux code can be re-generated in the original form, and processed as normal by Kconfig, cpp or make tools. It provides a two-way view to understand and maintain the kernel—one with state-of-art variability technique (i.e., cpp), and another using ART.

Like cpp, ART manipulates code in an unrestrictive way, with no concern for the syntactic or semantic rules of the underlying programming language. Freeing from language constraints makes ART powerful to represent the groups of similar structures of arbitrary kind differing in arbitrary ways in generic forms.

E. Trade-offs and Threats to Validity of Results

The flexibility of manipulating the code in unrestricted way comes at the price of not being able to quarantine the correctness of the generated code. Unrestrictive program manipulation decreases the type-safety of the program. Also, the trade-off is between the benefits and cost of learning the new technique. ART syntax is very simple and consists of only few constructs (such as *#adapt*, *#insert-break* mechanism, *#while*). Yet, building quality ART templates require skilled experts, so that the benefits of ART outweigh the burden of learning and adopting it.

The benefit of ART depends on the degree of redundancy in a software system that cannot be fixed by simple refactoring. The bigger the size of software systems, the higher the likelihood of redundancies and evolutionary changes, and hence more will be the benefits of using ART. It follows that families of similar systems should be prime candidates for ART template views, as there is much similarity among components of such systems. Thus, the proposed technique seems to have more direct relevance in the SPL context, where we have the role of domain engineer who is responsible for building reuse-based productivity solutions that serve many systems in long run. ART templates belong to that category of solutions.

VIII. Related Work

We discuss related work on cloning in the Linux kernel and techniques that help programmers achieve non-redundancy, including XVCL (the predecessor of ART).

A. Cloning in the Linux Kernel

Cloning in the Linux kernel has been extensively studied in the literature [15][16][17][21][24] mainly focusing on the detection of small cloned code fragments. In the Linux v2.4.0, Casazza et al. [17] reported cloning of 15.5% between arch and drivers subsystems. They also reported 13.6% cloning between arch and kernel subsystems. Other study showed that file subsystem had 12% clone coverage in the Linux v2.4.19 [16]. In the Linux v2.6.37.6, 8% code similarity between drivers (/sound and /drivers directories) has been reported [15]. An empirical study of cloning among SCSI drivers is done by [24]. We found these cloning rates to be lower than those reported in similar studies for web applications [19] (60%-90%) or class libraries [7] (68%).

Compared to other studies, this paper aims at the detection and analysis of big clones instead of small cloned code fragments that are detected and analyzed by other studies.

B. Managing Redundancies in Software Systems

Simple-minded development often leads to cloning (copy-paste-modify practice). As mentioned earlier, cloning may also be done for good reasons [5]. Still, non-redundancy has been always considered an important quality of well-designed software. The Software Engineering principle of generality encourages avoiding repetitions and building parameterized software solutions that can be reused in many contexts. Macros were an early attempt to make programs adaptable to various contexts. Goguen popularized the ideas of parameterized programming [25]. Among programming language features, type parameterization [12] (called generics in Ada, Eiffel, Java and C#, and templates in C++), higher-order functions, and inheritance can help avoid repetitions in certain situations. Design techniques such as iterators, design patterns, table-driven design (e.g., in compiler-compilers), and modularization with information hiding are supportive in building generic programs. The Standard Template Library (STL) is a premier example of engineering benefits gained by generality [26]. Techniques have also been proposed to lift sufficient code similarity from the code to the architectural level [27][28].

ART uses templates and code generation to achieve non-redundancy. ART templates can represent any groups of clones (e.g., files or directories) with arbitrary differences among them (as opposed to only type-parametric differences in C++ templates or Java generics).

C. ART versus cpp + scripts and XVCL

One can also achieve non-redundancy by parameterizing and wrapping the code with cpp, shell scripts, and make files. An example of that can be found in the JDK buffer library described in [7]. SUN developers used cpp, scripts, and make files to build a non-redundant representation from which actual buffer classes are derived. A quick inspection of the code reveals that such representation may serve only its author and cannot be considered as a viable method to engineer programs.

Sample ART templates shown in the paper may also look complex. At the first glance they do. But, the fact is that ART is governed by only five important constructs (i.e., *#adapt*, *#output*, *#insert-break* mechanism, *#while*, and *#select*) that are neatly integrated to form a method that can be learned easily. Experience with XVCL (the ART predecessor) demonstrates that large code can be effectively managed achieving non-redundancy in the program areas where it matters [19]. Despite user-defined syntax, ART further

improves the user experience by providing the following improvements to XVCL:

Easy to learn: XVCL is a dialect of XML and uses XML trees and a parser for processing. ART parts with XML syntax and processing. It offers a cpp based flexible and more readable user-defined syntax. Just because cpp is so widely used, learning ART is not so tedious.

More generalized: Contrary to XVCL, developers can easily blend ART with the programming technologies of their choice. It is because the developer can define their own syntax and hence can avoid conflicts with the base languages.

Expanding the customization options under #adapt command: In XVCL, the only command that you can place under *adapt* is *insert*. ART allows to use any command under *#adapt*. We found using *#set*, *#while* and *#select* commands under *#adapt* to be particularly very useful.

Robust structure instead of unreadable loops: In XVCL, *while* loops using many multi-value variables can be quite confusing. ART introduces a structure called *set-loop* which gives the possibility to store and use more multi-value variables together as one loop descriptor data structure.

More flexible: ART is more flexible than XVCL, as it allows the adaptation of a file even though the file might not contain any ART commands. Such adaptation would simply copy the adapted file to the output stream.

D. Comparison with Other Techniques

Companies today often develop and maintain custom versions of the same software system for different customers using SPL [4]. The core idea is to manage the system family as a whole from a base of core assets designed for ease of adaptation in various reuse contexts.

In the SPL context, ART attempts to capture and streamline the end-to-end process of adapting software from the specifications of variant features (e.g., in Linux called configuration options) to the architectural structures and the code. ART templates can manipulate any textual file independent of their contents. So, it can also manage variability in documentation and test cases, keeping all textual SPL core assets in sync with evolving code.

Techniques proposed in research to manage variability in SPL are mostly based on the principle of separation of concerns (SoC), introduced by Dijkstra in early 1980's [29]. The goal of SoC is to deal with concerns one by one, independently from other concerns. When applied at the level of design and implementation, SoC attempts to compose software from components implementing different concerns. Concerns that nicely fit into conventional modules are easy to deal with. The challenge is to tackle cross-cutting concerns that are tightly coupled with the rest of a program, and cannot be easily modularized in a conventional way. There have been attempts to bring SoC down to the design and implementation levels. Aspect-oriented programming (AOP) [30], multi-dimensional SoC (MDSOC) from IBM [31], feature-oriented programming (FOP) [32], and colored IDE (CIDE) [33] are among the most widely published of such techniques. Among these techniques, AOP has been widely used. In AOP, various computational aspects are programmed separately and weaved at specified join points into the base program. AOP can separate a range of programming aspects such as synchronization, persistence, security transaction management, authentication/authorization, and others. Separated aspects can be easily modified and added/deleted to/from program modules. Because of that, a number of authors proposed AOP as a variability technique in the SPL. A study to test this hypothesis revealed difficulties in using AspectJ to deal with features that have chaotic impact on the base code [34]. While AOP deals with big chunks of functionalities (i.e., aspects) reasonably, it lacks a mechanism to handle variations at the lower levels of granularity. ART on the other hand, can handle variations at any levels of granularity. Walkingshaw et al. [35] provided a systematic and broader perspective on variational data structures. Properties related to program customizations are encapsulated in these variability-aware data structures.

IX. Conclusions

A study of industrial systems has shown that around 50% of small cloned code fragments tend to be contained in big clones [36]. While big clones are certainly intentional, they contribute to the increased program size and complexity. Therefore, big clones create a useful window from which to understand and manage clones at all levels of granularity.

In this paper, we presented a technique for managing big clones with non-redundant templates built with ART. ART templates manage big clones without conflicts with programming languages and other techniques used for managing variability. We demonstrated the technique with examples from the Linux kernel that uses cpp (among other techniques) to manage variability.

In various similarity groups, by unifying clones into non-redundant templates, ART eliminated 30-50% of the code. Non-redundant views revealed by ART templates improve program understanding. Program relations that have to do with the impact of changes are important in program understanding, maintenance and evolution, but remain mostly implicit in conventional programs. ART templates expose and explicate some of these program relations. For example, when maintaining duplicated code we often must know where such duplicates are and how they are different, in order to decide if and how each of them should be modified. ART makes such information more visible and tractable, reducing the risk of unexpected errors when changing programs.

ART blends without conflicts with the underlying programming language and any other techniques used to manage variability in a software system. Therefore, we can use ART to handle big clones, while other techniques (e.g., cpp and Kconfig in the Linux kernel) deal with other aspects of the overall variability management problem. Such seamless integration is necessary to allow the developers to painlessly inject ART templates into the projects in mature stages of evolution when big clones start emerging. ART syntax is user-defined to make such injection easy, without

affecting already existing software solutions and people who work with them. In the Linux context, ART can be viewed as an extension of cpp where ART commands syntactically resemble cpp directives, and can be incrementally learned as extensions that enhance reuse capabilities of cpp. The Linux Developers work with ART templates of the program, while the ART Processor generates the Linux code in its original form for the Linux SysAdmins.

Any new technique brings some overhead, requires learning and skillful application. ART is no different from other techniques in this respect. ART templates are not created for quick gains during development, but for long-term gains during software evolution and reuse. ART aims to benefit long-lived systems that undergo extensive evolutionary changes, or need to be tailored to the needs of multiple customers.

Acknowledgment

We are thankful to Ulf Pettersson, Technical Director, Info-Software Systems, ST Electronics Pte. Ltd., Singapore for applying ART in his projects and providing us with invaluable feedback.

References

[1] P. Clements and D. Muthig, (Editors) Proceedings Workshop on Variability Management–Working with Variation mechanisms, in *SPLC*, 2006, IESE-Report No 152.06/E Version 1.0, Germany, October 15, 2006

[2] C. L. Goues, S. Forrest, and W. Weimer, "The case for software evolution," in *FoSER,* 2010, pp. 205–210, http://dx.doi.org/10.1145/1882362.1882406

[3] R. Koschke, "Identifying and removing software clones", in *Software Evolution*, Springer Berlin Heidelberg, 2008, pp. 15–36, http://dx.doi.org/10.1007/978-3-540-76440-3_2

[4] P. Clements and L. Northrop, *Software product lines: practices and patterns*. Addition-Wesley, 2002

[5] C. Kapser and M. W. Godfrey, ""Cloning considered harmful" considered harmful," in *WCRE*, 2006, pp. 19–28, http://dx.doi.org/10.1007/s10664-008-9076-6

[6] G. P. Krishnan and N. Tsantalis, "Unification and refactoring of clones," in *CSMR-WCRE*, 2014, pp. 104–113, http://dx.doi.org/10.1109/CSMR-WCRE.2014.6747160

[7] S. Jarzabek and L. Shubiao, "Eliminating redundancies with a "composition with adaptation" meta-programming technique," in *ESEC/FSE*, 2003, pp. 237–246, http://dx.doi.org/10.1145/949952.940104

[8] S. Schulze, S. Apel, and C. Kästner, "Code clones in feature-oriented software product lines," in *GPCE*, 2010, pp. 103–112, http://dx.doi.org/10.1145/1942788.1868310

[9] R. Koschke, "Frontiers of software clone management," in *FoSM*, 2008, pp. 119–128, http://dx.doi.org/10.1109/FOSM.2008.4659255

[10] Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker, and K. Czarnecki, "An exploratory study of cloning in industrial software product lines," in *CSMR*, 2013, pp. 25–34, http://dx.doi.org/10.1109/CSMR.2013.13

[11] R. Lotufo, S. She, T. Berger, K. Czarnecki, and A.Wąsowski, "Evolution of the Linux kernel variability model," in *SPLC*, 2010, pp. 136–150, http://dx.doi.org/10.1007/978-3-642-15579-6_10

[12] R. Garcia, J. Jarvi, A. Lumsdaine, J. G. Siek, and J. Willcock, "A comparative study of language support for generic programming," in *OOPSLA*, 2003, pp. 115–134, http://dx.doi.org/10.1145/949305.949317

[13] S. Jarzabek, P. Bassett, H. Zhang, and W. Zhang, "XVCL: XML-based variant configuration language", in *ICSE*, 2003, pp. 810–811, http://dx.doi.org/10.1109/ICSE.2003.1201298

[14] P. Ye, X. Peng, Y. Xue, and S. Jarzabek, "A case study of variation mechanism in an industrial product line," in *ICSR*, 2009, pp. 126–136, http://dx.doi.org/10.1007/978-3-642-04211-9_13

[15] A. Kadav and M. M. Swift, "Understanding modern device drivers," in *ASPLOS*, 2012, pp. 87–98, http://dx.doi.org/10.1145/2150976.2150987

[16] C. Kapser and M. W. Godfrey, "Toward a taxonomy of clones in source code: A case study," in *ELISA*, 2003, pp. 67–78

[17] G. Casazza, G. Antoniol, U. Villano, E. Merlo, and M. Di Penta, "Identifying clones in the Linux kernel," in *SCAM*, 2001, pp. 90–97, http://dx.doi.org/10.1109/SCAM.2001.972670

[18] H. A. Basit and S. Jarzabek, "A data mining approach for detecting higher-level clones in software," *IEEE Trans. Softw. Eng.*, vol. 35, no. 4, pp. 497–514, 2009, http://dx.doi.org/10.1109/TSE.2009.16

[19] U. Pettersson and S. Jarzabek, "Industrial experience with building a web portal product line using a lightweight, reactive approach," in *ESEC/FSE*, 2005, pp. 326–335, http://dx.doi.org/10.1145/1081706.1081758

[20] S. Jarzabek, U. Pettersson, and H. Zhang, "University-industry collaboration journey towards product lines," in *ICSR*, 2011, pp. 223–237, http://dx.doi.org/10.1007/978-3-642-21347-2_17

[21] S. Livieri, Y. Higo, M. Matsushita, K. Inoue, "Analysis of the Linux kernel evolution using code clone coverage," in *MSR*, 2007, pp. 22, http://dx.doi.org/10.1109/MSR.2007.1

[22] F. P. Brooks, Jr., "No silver bullet essence and accidents of software engineering," *IEEE Computer*, vol. 20, no. 4, pp. 10–19, 1987, http://dx.doi.org/10.1109/MC.1987.1663532

[23] M. Kim, V. Sazawal, D. Notkin, and G. Murphy, "An empirical study of code clone genealogies," in *ESEC/FSE*, 2005, pp. 187–196, http://dx.doi.org/10.1145/1081706.1081737

[24] W. Wei and M. W. Godfrey, "A study of cloning in the Linux SCSI drivers," in *SCAM*, 2011, pp. 95–104, http://dx.doi.org/10.1109/SCAM.2011.17

[25] J. A. Goguen, "Parameterized programming," *IEEE Trans. Softw. Eng.*, vol. SE-10, no. 5, pp. 528–543, 1984, http://dx.doi.org/10.1109/TSE.1984.5010277

[26] D. R. Musser, G. J. Derge, and A. Saini, *STL tutorial and reference guide: C++ programming with the standard template library*. Addison-Wesley Professional, 2009

[27] T. Mende, R. Koschke, and F. Beckwermert, "An evaluation of code similarity identification for the grow-and-prune model," *J. of Soft. Maint. & Evol.*, vol. 21, no. 2, pp. 143–169, 2009, http://dx.doi.org/10.1002/smr.402

[28] P. Frenzel, R. Koschke, A. P. J. Breu, and K. Angstmann, "Extending the reflexion method for consolidating software variants into product lines," in *WCRE*, 2007, pp. 160–169, http://dx.doi.org/10.1109/WCRE.2007.28

[29] E. W. Dijkstra, "On the role of scientific thought," in *Selected Writings on Computing: A Personal Perspective*, ed: Springer, 1982, pp. 60–66, http://dx.doi.org/10.1007/978-1-4612-5695-3_12

[30] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *ECOOP*, 1997, pp. 220–242, http://dx.doi.org/10.1007/BFb0053381

[31] P. Tarr, H. Ossher, W. Harrison, and S. Sutton, "N degrees of separation: multi-dimensional separation of concerns," in *ICSE*, 1999, pp. 107–119, http://dx.doi.org/10.1145/302405.302457

[32] D. Batory, J. N. Sarvela, and A. Rauschmayer, "Scaling step-wise refinement," *IEEE Trans. Soft. Eng.*, vol.30, no. 6, pp. 355–371, 2004, http://dx.doi.org/10.1109/TSE.2004.23

[33] C. Kästner, S. Apel, and M. Kuhlemann, "Granularity in software product lines," in *ICSE*, 2008, pp. 311–320, http://dx.doi.org/10.1145/1368088.1368131

[34] C. Kästner, S. Apel, and D. Batory, "A case study implementing features using AspectJ," in *SPLC*, 2007, pp. 223–232, http://dx.doi.org/10.1109/SPLC.2007.5

[35] E. Walkingshaw, C. Kästner, M. Erwig, S. Apel, and E. Bodden, "Variational data structures: exploring tradeoffs in computing with variability," in *ONWARD!*, 2014, pp. 213–226, http://dx.doi.org/10.1145/2661136.2661143

[36] H. A. Basit, U. Ali, S. Haque, and S. Jarzabek, "Things structural clones tell that simple clones don't," in ICSM, 2012, pp. 275–284, http://dx.doi.org/10.1109/ICSM.2012.6405283