Proactive Detection of Higher- Order Software Code Conflict's System

Godswill U. Nwamuruamu
University of Port-Harcourt
Rivers State, Nigeria

Laeticia N. Onyejegbu
University of Port-Harcourt
Rivers State, Nigeria

ABSTRACT
Collaborative development can be hampered when conflicts arise because developers have inconsistent copies of a shared project. We present an approach to help developers identify and resolve conflicts early, before those conflicts become severe and before relevant changes fade away in the developers' memories. A proactive high-order conflict detector helps programmers in a collaborative environment to detect conflicts and resolve same early to avoid malfunction of the software after deployment. With this, system conflicts are detected on time during design and resolved before they become more difficult to handle or before the code becomes too voluminous to debug. Using Java as a design tool the system was developed to detect code errors earlier and faster than already existing systems. The result obtained shows that the system resolves and detects conflicts early enough to avoid damage to the design in record time. The system designed uses less memory space with highly effective software activity which maximizes the host system resources. The methodology adopted for this design is the object oriented approach which gives a lot of avenues for conflict resolution and encourages code flexibility.

Keywords
Proactive, Conflicts, FLAME, Conflict detection, Software model high- order.

1. INTRODUCTION
Modern application techniques are often huge and complicated, demanding several technological innovation groups that include of a variety of members for their growth. During the growth, numerous choices are created on various elements of the program under growth such as the dwelling, the features, as well as the non-functional qualities of the program such as efficiency, security, etc. This essential growth activity of selection is known as application style. A key set of stakeholders who interact with in application style, application designers, create style choices that determine the dwelling of the program, reify those choices into application designs and develop the designs together [1].

The groups of application designers, when they style a huge application program, often split the program into flip subsystems, at one time styles each of those, and combine them later. A variety of style surroundings appeared to support this collaborative procedure for application design progress. There are the group publishers offering an allocated whiteboard" or connect the designs in real-time but the significant research attempt has been toward the asynchronous, copy-edit-merge style application design edition control techniques (VCSs). Those VCSs offer each designer her individual workplace by generally syncing the designs in an on-demand style to parallelize the architects' work and increase their efficiency.

However, the reduce synchronization of the VCSs reveals application designers to the chance of creating style choices that issue with each other, known as style disputes. In general, style disputes can be classified into two different types: synchronization and higher-order disputes. A synchronization issue is a set of contrary modeling changes by several designers created to the same doll or to carefully related relics, which means they cannot be combined together. A higher-order issue is a set of modeling changes by several designers that can be combined but together breach the body reliability guidelines (e.g., cardinality based on the meta-model). While both kinds cause similar threats, they vary in that they require different sets of recognition techniques.

Design disputes are a significant task in collaborative application style. Today's VCSs identify disputes only when application designers connect their designs. As a result, the designers often create changes to the design without fully understanding what issues may occur when they combine their own changes with the others' changes. It is also possible that new changes created after an issue has been presented need to be changed in the procedure for solving the issue, which results in lost persistence. Moreover, the current pattern of worldwide application technological innovation, in which application technological innovation groups tend to be allocated geographically due to economic advantages only worsens the procedure by decreasing the possibilities for direct interaction among the designer.

In application environment, there are some disputes that surface and need to be fixed. These disputes are: Lack of ability to identify issue at the start of a practical style, that is, before an application professional syncs her design and lastly becomes aware of them: This thesis will present a technique that relieves the chance of having style disputes by proactively discovering them and telling the application professional of the issue information beginning.

The aim of this research is to develop a Proactive Detection of Higher-order Software code Conflict's system that can identify errors in codes. The specific objectives are to style an enhanced design of Proactive Detection of Higher-order Software Design Conflicts, to apply the design using free coffee development language and to evaluate the design with current practical design

2. REVIEW OF RELATED WORK
Software Technological innovation is a division of data technology that is focused on how to build software techniques that are good, useful and efficient. It is also a self-discipline whose aim is to generate fault-free software that meets the user's needs and is provided promptly within budget. The objectives of implementing an engineering technique are to build a program that is efficient, easy to understand, affordable, convenient and recyclable [2]. A software program is efficient if it works properly without failing. The progress in software engineering is all about

changes. Program engineering involves techniques, often controlled by a program growth procedure, within the purpose of helping the stability and maintainability of software techniques [3]. Software engineering therefore is a critical area in which nations must spend intensely on, to make their economic growth a reality. [4] **Wloka et al (2009)** worked on "Safe-Commit Analysis to Facilitate Team Software Development," which they used proactive conflict detection tools to perform deeper analyses such as compilation, unit testing, and so on. Safe-commit proactively identifies "committable" changes that will not make test cases fail by running them in the background. [5] **Brun et al (2011)** worked on "Crystal: Precise and Unobtrusive Conflict Warnings, they used two tools in this group, Crystal and We-Code , which are used proactively to perform merging, compilation, and testing of new changes developers make to source code in the background and notify the developers if any of the steps fails. [6] **Algert and Watson (2002)** worked on Conflict management and introduced it for individuals and organizations to identify the danger involved when conflict arises and the little solution to solve it in a modern way. [7] **Labib et al (2009),** introduced a practice called Early User Interface Development' (EUID) for agile software methodologies. In the EUID model, a GUI is designed at an early stage of agile iteration, developed and presented to the customer for feedback before starting a new iteration with a set of new requirements. This feedback mechanism is not only for the look of the GUI design but also for the behavior of using the GUI. In this way designers and customers can communicate and produce the actual required user interface (UI). [8] **Leffingwell (2007)** discusses which agile practices scale to large systems development and report on his experience of using an agile approach to develop a large medical system with 300 developers working in geographically distributed teams. Large software system development is different from small system development in a number of ways:

1. Large systems are usually collections of separate, communicating systems, where separate teams develop each system. Frequently, these teams are working in different places, sometimes in different time zones. It is practically impossible for each team to have a view of the whole system. Consequently, their priorities are usually to complete their part of the system without regard for wider systems issues.

2. Large systems are 'brownfield systems'; that is they include and interact with a number of existing systems. Many of the system requirements are concerned with this interaction and so don't really lend themselves to flexibility and incremental development. Political issues can also be significant here—often the easiest solution to a problem is to change an existing system. However, this requires negotiation with the managers of that system to convince them that the changes can be implemented without risk to the system's operation.

3. Where several systems are integrated to create a system, a significant fraction of the development is concerned with system configuration rather than original code development. This is not necessarily compatible with incremental development and frequent system integration.

4. Large systems and their development processes are often constrained by external rules and regulations limiting the way that they can be developed, that require certain types of system documentation to be produced, etc.

5. Large systems have a long procurement and development time. It is difficult to maintain coherent teams who know about the system over that period as, inevitably, people move on to other jobs and projects.

6. Large systems usually have a diverse set of stakeholders. For example, nurses and administrators may be the end-users of a medical system but senior medical staff, hospital managers, etc. are also stakeholders in the system. It is practically impossible to involve all of these different stakeholders in the development process

[9] **Guntamukkala et al (2006)** were able to identify canonical functions to help project manager select appropriate software development model for each potential or planned project. Their analysis of testing of data's led to three categories of software life cycle models and canonical functions that can be used by project managers to select the most suitable model for a given project situation. Their analysis also showed the perceptions of 74 participants – project managers, aout their preference of six software life cycle models varying in degree of flexibility, in different situations in the sense that every software project has its unique set of characteristics. A software life cycle model that is suitable for one project situation may not be suitable for another.

[10] **Walia and Khalid (2014)** worked on the "Impact of interpersonal conflict on requirements", and identified the five types of interpersonal conflicts which may lead to project failure as shown in case study of Enterprise Resource Planning system also identify the solution of attaining and achieving the solution towards the conflict.

2.1 Program Technological innovation Technique

All software growth techniques follows a venture schedule and must go through different levels during the growth life-cycle such as specifications collecting, research, style, execution and examining. It is essential for most application designers to follow some sort of software engineering methodology. The methodology is an important tools to control sources, product style and quality guarantee, in order to generate "well designed software". [11]

Software engineering strategies are essential frameworks that guide application designers to reach their last goal of getting the ultimate application, the choice of which methodology to use in an improvement venture is carefully related to the size of the applying program and the environment it ought to function.

2.2 Software Conflicts

A "conflict" happens when more than one processer is trying to connect to the same resource (a storage portion) at once. To prevent crucial competitions and inconsistency, only one processer (CPU) at a time is allowed to have access to a particular information framework (a storage portion). While other CPUs trying to have access at the same time are locked-out. [6].

Three situations can be recognized why this nonproductive wait around is necessary, practical, or not practical. The nonproductive wait around is necessary when the accessibility is to a prepared record for a low stage arranging function. The nonproductive wait around is not necessary but practical in the situation of an important area for synchronization/IPC functions, which require shorter period than a perspective switch (executing another procedure to avoid nonproductive wait). Idle wait around is instead not practical in situation of a kernel crucial area for device management, present in

monolithic popcorn kernels only. A microkernel instead drops on just the first two of the above situations.

In a multiprocessor program, most of the disputes are kernel-level disputes, due to the having accessibility to the kernel stage crucial sections, and thus the nonproductive wait around periods generated by them have an important impact in performance deterioration. This nonproductive wait around time boosts the average variety of nonproductive processor chips and thus reduces scalability and relative performance [12].

Conceptually, the most legitimate solution is to break down each kernel information framework in more compact separate substructures, having each a shorter elaboration time. This allows more than one CPU to connect to the original information framework.

Many uniprocessor techniques with ordered protection websites have been approximated to spend up to 50% of plenty of your energy performing "supervisor mode" functions. If such techniques were tailored for multiprocessing by setting a lock at any having accessibility to "supervisor state", L/E would easily be greater than 1, becoming a program with the same data transfer usage of the uniprocessor despite the quantity of CPUs.

3. METHODOLOGY

We will present situation study and a style that will create a Practical Recognition of Higher-order Application Style Disputes that can identify oblique issue in a development requirements of a program . Ideal Options Growth and Research (SODA) is a method for working on complicated issues. It is an approach designed to help professionals help their potential customers with unpleasant issues.

SoDA, is a general software methodology for creating software whose outputted source rule is an argumentation concept for the issue at hand. This system describes a advanced level procedure demanding from the designer to consider questions about the specifications of the issue at various circumstances without the need to consider the actual software rule that will be produced. Application is thus designed in a principled way with high-level declarative exe rule.

Using this methodology, our development techniques will contains techniques and the perform group necessary at each level of the work version and identify issue previously and way to resolve them.

To relieve the risk, the proposed product is used to uncover unnoticed higher-order style disputes, we have developed an extensible, and function centered collaborative application style structure, known as Framework for Signing and Examining Modeling Events (FLAME) and IDE which functions as resources for ongoing consolidating within the IDE FLAME decreases the period of time during which the disputes can be found but unknown to application designers by proactively performing the issue recognition activity that includes a trial consolidating of modeling changes and performance of reliability checking resources in the. FLAME consequently provides the issue details to the designers in case the architects' attention is required.

FLAME has two features that differentiate it from the current practical issue recognition resources.

1. FLAME and IDE is extensible. Software modeling surroundings vary in their modeling resources, 'languages', and the suitable reliability pieces. FLAME uses an event-based structure in which highly-decoupled elements exchange messages via implied invocation, allowing flexible program structure and variation. FLAME uses this event-based structure to provide precise expansion points for connecting a variety of off-the-shelf resources, namely, modeling resources and issue recognition engines that are most appropriate for the given modeling atmosphere.

2. FLAME and IDE is operation-based. It syncs the models at the granularity of a single modeling function such as creation, upgrade, or removal of a modeling element.

The IDE using coffee development language is the better and a novel way to provide an alternative that decreases the amount of data designers have to process. The existing remedy consistently combines uncommitted and committed changes to create a background program that is examined, collected, and examined to identify disputes with high perfection and precision on part of designers, while they are development, that is, before check-in. Recognized disputes are then provided to the affected designers inside the IDE.

In evaluation to the current program this details the progress of our solution, provides our full-fledged tool, and its scientific assessment using managed user tests.

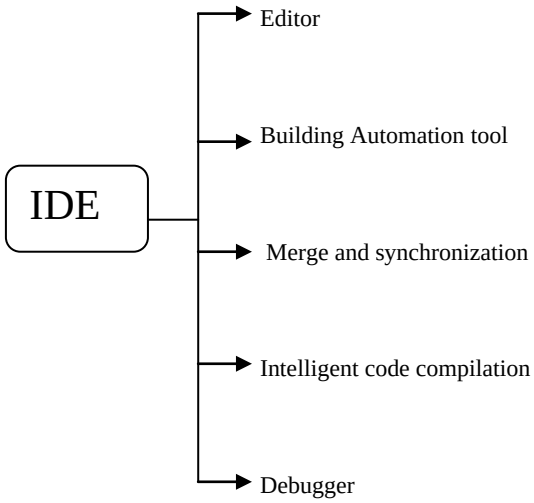

Fig 1: Components of IDE

3.1 Explanation of IDE

IDE: is illustrated as integrated development environment (IDE), it is a software application that provides comprehensive facilities to computer programmers for software development. An integrated development environment normally consists of a source code editor, build automation tools and a debugger. Most modern IDEs have intelligent code completion. The IDEs, such as NetBeans and Eclipse, contain a compiler, interpreter, or both; others, such as Sharp Develop and Lazarus, do not. The boundary between an integrated development environment and other parts of the broader *software development environment* is not well-defined.

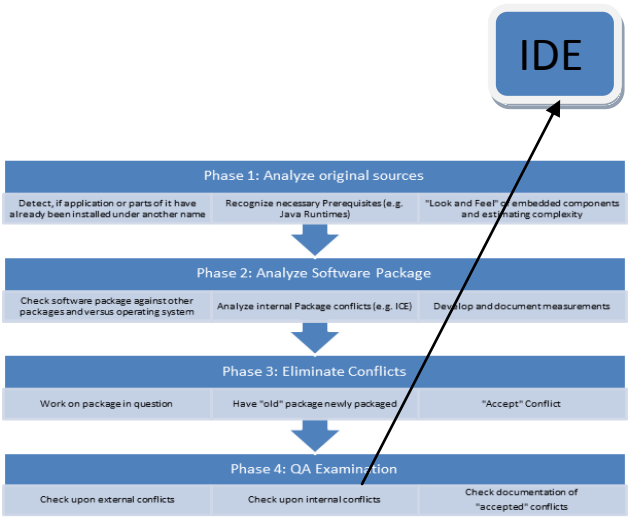

Fig 2: High-Level model of the Proposed System

3.2 System Requirement

The running of this method and the execution of the task management software system demand effective pc with the following minimum requirement: A processer machine of Pentium IV standard with 1GHz speed it will have memory (RAM) dimension 1GB or above and a video visual adaptor (VGA) screen with 32bit high color or above with a Hard Disk size of not less than 1GB.

For versatility it will use a Window operating system and will be powered by a JavaScript Allowed Browser of Internet Traveler 7 and above or Firefox 2 and above (JDK 8.01 version) and Net Beans 8.0 edition.

3.3 Implementation plan

Proactive detection of high order code conflicts application implementation plan:

Phase 1: Install software on the main drive

Phase 2: Link every fill directory on the system

Phase 3: always specify the coding language and environment

Phase 4: Run the Model on the growth code

Phase 4: Copy report and delete /resolve detected conflicts

Phase 5: Update the growth system code before deployment;.

Fig 3: Home page of the Design

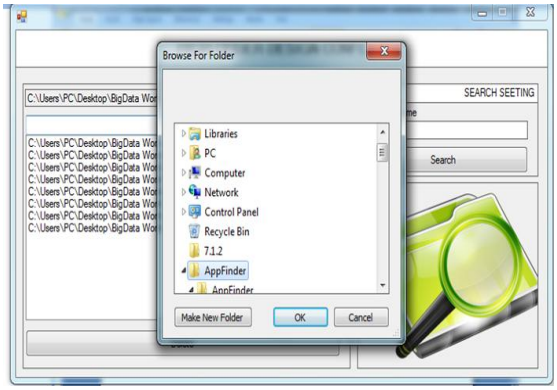

Fig 4: searching for folders to scan

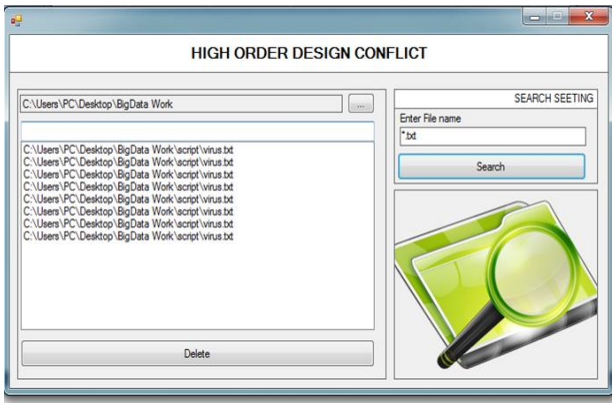

Fig 5: Conflicting files detected

3.4 Cost Benefit Analysis

With the implementation of this model the output of design works will be with reduced error and this reduces cost in diverse forms. Memory wise it will reduce the amount of unuseable codes in the system since the incorrect and unusded codes wil be detected in the system and deleted. It also rduces the issue of developing a system that will be faulty and therefore cannot perform effectively due to error. This often leads to redesign or condemn of the prototype.

4. RESULTS AND DISCUSSION

The performance evaluation of an application package is calculated with the matrices used. Several matrices can be used in the evaluation of an application package ranging from speed, time, efficiency, memory consumption amount of energy consumed and so on. The proposed system was tested alongside the existing prototype and the following results were obtained.

Table 1 Through-put for software performance

Computers	Existing System	Proposed System
1	30min	10min
2	40min	25min
3	60min	30min
4	45min	30min

From the above table the proposed system was tested against the existing systems in four different systems and the proposed system evidently used less time for processing saving time and energy.

Also when the system usage was tested for other functionalities like communication level with other programs, amount of energy consumed number of operations perform effectively the chart in (Fig. 6) clearly demonstrates that even with the same time duration the proactive system runs more operations , communicates better while consuming less energy and memory.

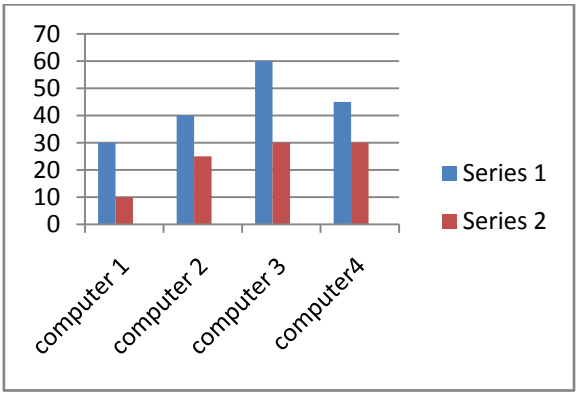

Fig 6: Comparison of Existing and Proposed System

5. CONCLUSION

Speculative analysis over version control operations provides precise information about pending conflicts between collaborating team members. These pending conflicts — including textual, build, and test — are guaranteed to occur (unless a developer modifies a committed change). Learning about them earlier allows developers to make better-informed decisions about how to proceed, whether it be to perform a safe merge, to publish a safe change, to quickly address a new conflict, to interact with another developer etc.

Our retrospective, quantitative study of over 550,000 development versions of nine open-source systems, spanning 3.4 million distinct (and a total of over 500 billion, over all versions) NCSL, confirms that:

a. Conflicts are the norm rather than the exception,

b. That 16% of all merges required human effort to resolve textual conflicts,

c. That 33% of merges that were reported to contain no textual conflicts by the VCS in fact contained higher-order conflicts, and

d. Those conflicts persist, on average, for 10 days (with a median conflict persisting 1.6 days).

Although there is a significant amount of qualitative and anecdotal evidence consistent with our findings, the only previous quantitative research we could find was Zimmermann's work. We expanded on his work in several dimensions.

The essence of this study is to reduce software code error which has derided our society of good and effective software usage and development. To do that we had to create a means to reduce one of the most evident tools that causes ineffectiveness which is conflicts and further more we looked into those conflicts that surfaces after the design has been concluded "high-order". By this study developers can freely develop an error free software by resolving conflicts in codes even before they become evident.

With this software collaborative design can easily be synchronized and error detected if any, reducing time wastage on troubleshooting and debugging.

6. RECOMMENDATION

It is recommended that program designers from all around the world should eliminate code errors and application mistakes before implementation and release to the general public.

7. REFERENCES

[1] Gilb, T., "(1998): Evolutionary Development. ACM. Software Engineering, (1), 17-23.

[2] Highsmith, J. A. (2002). Adaptive Software Development: A Collaborative Approach to Managing Complex Systems. New York: Dorset House. 1(1), 5-8.

[3] Nwachukwu, E.O. and Eke, B.O.(2008). Critical Analysis of Software Development Strategies. Journal of Science and Technology. 7(4), 1-7

[4] Wloka J., B. Ryder, F. Tip, and X. Ren, (2009) "Safe-Commit Analysis to Facilitate Team Software Development," in Proceedings of the 31st International Conference on Software Engineering (ICSE) IEEE Computer Society. (7),507–517

[5] Brun, R. Holmes, M. D. Ernst, and D. Notkin (2011): "Crystal: Precise and Unobtrusive Conflict Warnings," in Proceedings of the 8th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE). ACM, (12), 444–447.

[6] Algert, N.E., and Watson, K. (2002): Conflict management: introductions for individuals and organizations. Bryan, TX: (97)775-870

[7] Labib, C., Hasanein, E., And Hegazy, O., (2009).Early Development Of Graphical User Interface (GUI) In Agile Methodologies. Journal Of Computation Methods In Sciences And Engineering. (9), 239-249.

[8] Leffingwell, D. (2007). *Scaling Software Agility: Best Practices for Large Enterprises.* Boston: Addison-Wesley. **37** (12), 26–34.

[9] Guntamukala, V., Wen, H. J and Tarn, J.M. (2006).An empirical study of selecting software development life cycle models. Human systems management. 25(1), 265-278.

[10] Walia, Khalid Shergil, (2014): "Impact of interpersonal conflict on requirements", A Research Review", University of westernontario , 1(2), 10-19.

[11] Royce, W.W. (1970). Managing the Development Of Large Software Systems: Concept And Techniques, Proceeding WESCON. (2), 23-55

[12] Madnick, Stuart Elliot (1968) *Multi-processor software lockout* Proceedings of the 1968 23rd ACM national conference, (1), 19 – 24.