Software reuse: A survey

Tajinder Singh Sondhi, Shivam Ghildyal, Srishti Sabharwal, Akash Nagarkar, K Lavanya
SCOPE, VIT University, Vellore, India

ABSTRACT
Software reuse is an important but hidden part of software industries and it provides them with predefined software components in order to build better software, more quickly and at a lower cost. Reusing software components helps in an accelerated, risk free and cost-effective development of software, but over few decades many issues have arrived. We have studied numerous methodologies available for software reuse starting from the traditional ones like REBOOT, faceted implementation to the more recent ones like the PULSE methodology, object oriented methodology and design science research methodologies, pragmatic approach and maturity model. These methodologies have constantly provided better techniques that have enhanced the development time, reduced the development costs, decreased the process risk, improved its dependability and have provided a standardised approach for software reuse. In this paper, we have tried to compare and analyse the various techniques and methods available for solving the various problems associated with the reuse of software components.

Keywords *- Software reuse; plagiarism; stability; security; reuse techniques; reuse models*

1. Introduction
Slowing advancements in software productivity and increased demand means that most software organisations increasingly find themselves in the midst of a software crisis that inhibits their ability to produce manageable, high quality, cost-effective software. Software reuse is seen as one solution to reducing the time-to-market, and potentially improving the quality and productivity of software development.

Early attempts to attain reuse were founded on the development of general software libraries. This metaphor has inspired several researchers to investigate various retrieval methods based on faceted classification, keywords or fee-text documents retrieval. One of the difficulties is that software modules have to interact with each other; they need to be designed to be compatible with the systems that intend to reuse them. Implicit assumptions about the requirements may produce components that cannot be used as they conflict with the systems that are required to reuse them. The potential time-to-market benefits associated with reuse can be lost if excessive adaptation and subsequent regression, testing are required. However, recently approaches to library methods that include software kits, task neutral problem solving generic building blocks, component based software engineering and many others have had more success. We shall describe and review few of them in our survey.

2. Literature Survey
Standish et al., [Programming Environment Project, 1] discussed briefly some economic incentives for developing effective software reuse technology and notes that different kinds of software reuse, such as direct use without modification and reuse of abstract software modules after refinement have different technological implications. It sketches some problem areas to be addressed if we are to achieve the goal of devising practical software reuse systems. These include information retrieval problems and finding effective methods to aid us in understanding how programs work. There is a philosophical epilogue which stresses the importance of having realistic expectations about the benefits of software reuse.

Jones et al., [Reusability in Programming: A Survey of the State of the Art, 2] published a report which addresses the 1984 state of the art in the domains of reusable data, reusable architectures, reusable design, common systems, reusable programs, and reusable modules or subroutines. If current trends toward reusability continue, the amount of reused logic and reused code in commercial programming systems may approach 50 percent by 1990.

Neighbors et al., [The Draco Approach to Constructing Software from Reusable Components, 3] researched in which they discussed an approach called Draco to the construction of software systems from reusable software parts. In particular we are concerned with the reuse of analysis and design information in addition to programming language code. The goal of the work on Draco has been to increase the productivity of software specialists in the construction of similar systems. The particular approach we have taken is to organize reusable software components by problem area or domain.

Kernighan et al., [The Unix System and Software

Reusability, 4] discussed about the Unix system which contains a variety of facilities that enhance the reuse of software. These vary from the utterly conventional, such as function libraries, to basic architectural mechanisms, such as the Unix pipe. The Unix pipe, which makes whole programs building blocks of larger computational structures, has been the primary reason for the development of a literature of useful, but specialized programs-programs that would be too costly to write in a conventional programming language such as C.

Jones et al., [Issues in software reusability, 5] decided that the most productive use of the little time available to the group was to clearly determine the issues in software reusability and make recommendations wherever possible.

McCain et al., [Reusable software component construction - A product-oriented paradigm, 6] researched that software productivity advances are not keeping pace with increasing software demands. The cost of software systems development is increasing rapidly. Major contributing factors are lack of planned accommodation of future needs and reuse and the lack of "human engineering" in an initial software system development.

Anderson et al., [Reusable Software – A Mission Critical Case Study, 7] found that the cost of mission critical software is rising at an alarming rate. With the advent of a standard high order language (HOL), Ada, the concept of reusable software can now be realistically pursued. Incentives of reuse are discussed, along with recommendations for future study topics.

Honiden et al., [Software Prototyping With Reusable Components. Recently, 8] researched that a prototyping method has attracted attention as a software specification method. Though many methods have been proposed, no standard method has been established. This paper proposes a prototyping method with reusable components based on knowledge engineering.

Polster et al., [Reuse of software through generation of partial systems. 9] considers the problem of constructing partial systems, where the program of a partial system is obtained by selecting only those code segments of the complete program that implement the capabilities needed. A heuristic for determining fragments of a program system, which can serve as the building blocks for the programs of partial systems, is presented.

Frakes et al., [Software reuse through information retrieval, 10] researched that there is widespread need for safe, verifiable, efficient, and reliable software that can be delivered in a timely manner. Software reuse can make a valuable contribution toward this goal by increasing programmer productivity and software quality. Unfortunately, the amount of software reuse currently done is quite small. DeMarco [1] estimates that in the average software development environment only about five percent of code is reused.

Prieto-Diaz et al., [Implementing faceted classification for software reuse, 11] published an article whose central component is a software reuse library organized around a faceted classification scheme. The system supports search and retrieval of reusable components and librarian functions such as cataloging and classification.

Prieto-Díaz et al., [Making software reuse work: an implementation model, 12] gave an approach which is practical, effective, and has potential to make reuse a regular practice in the software development process.

Henninger et al., [Information access tools for software reuse, 13] gave an approach for investigating with a retrieval tool, named CodeFinder, which supports the process of retrieving software components when information needs are ill-defined and users are not familiar with vocabulary used in the repository.

Krueger [Software reuse, 14] gave a generic idea about importance of software reuse.

Isoda [Experience report on software reuse project: its structure, activities, and statistical results, 15] describes a four-year experimental software reuse project conducted at Software Laboratories, Nippon Telegraph and Telephone. The targets of reuse are program code modules stored in a common library.

Chen et al., [On the study of software reuse using reusable C++ components, 16] presented a way (based on object interface) to manufacture reusable software components and propose a software construction method with guidelines for using designed reusable C++ components.

Constantopoulos et al., [The software information base: A server for reuse, 17] presented an experimental software repository system that provides organization, storage, management, and access facilities for reusable software components.

Sindre et al., [The REBOOT approach to software reuse, 18] presented the REBOOT approach to software reuse, covering both organizational and technical aspects and the experiences so far from the applications.

Henninger et al., [An evolutionary approach to constructing effective software reuses repositories, 19] researched an approach that avoids these problems by choosing a retrieval method that utilizes minimal repository structure to effectively support the process of

finding software components.

Leach et al., [Software Reuse: methods, models, and costs, 20] describes software engineering in the form of modules.

Rine et al., [Investments in reusable software: A study of software reuse investment success factors, 21] researched the thesis that there is a set of success factors which are common across organizations and have some predictability relationships to software reuse. For completeness, this research also investigated to see if software reuse had a predictive relationship to productivity and quality.

Kim et al., [Software reuse: survey and research directions, 22] surveyed existing research on software reuse using a framework that encompasses a broad range of technical and nontechnical issues.

Kovacs et al., [Application of software reuse and object-oriented methodologies for the modelling and control of manufacturing systems, 23] drafted a paper on object-oriented design and the application of software reuse of components of flexible manufacturing systems FMS will be introduced. The goal of this research was to provide methods and tools to build up FMS simulation models and control strategies easily, fast and reliably.

Bayer et al., [PuLSE: a methodology to develop software product lines, 24] developed the PuLSETM (&oduct Line software Engineering) methodology for the purpose of enabling the conception and deployment of software product lines within a large variety of enterprise contexts.

Séméria et al., [Methodology for hardware/software co-verification in C/C++, 25] did efficient mapping of C/ C++ functional descriptions directly into hardware and software.

Gibb et al., [The integration of information retrieval techniques within a software reuse environment, 26] described the development of an information retrieval (IR) model for the indexing, storage and retrieval of documents created in extensible mark-up language (XML).

Sherif et al., [Barriers to adoption of software reuse: a qualitative study, 27] researched the barriers that occur at both the individual and organizational level, and suggested that those at the individual level are actually a consequence of the interaction of barriers caused at the organizational level.

Bin et al., [A research on open CNC system based on architecture/component software reuse technology, 28] researched on open CNC system based the current development situation of open computer numerical control (CNC) system and architecture/ component software reuse technology.

Sherif et al., [Managing peer-to-peer conflicts in disruptive information technology innovations: the case of software reuse, 29] examined the case of software reuse as a disruptive information technology innovation (i.e., one that requires changes in the architecture of work processes) in software development organizations.

Peffers et al., [A design science research methodology for information systems research, 30] proposed and developed a design science research methodology (DSRM) for the production and presentation of design science (DS) research in information systems (IS).

Garcia et al., [Towards a Maturity Model for a Reuse Incremental Adoption, 31] found that the software reuse practices have mostly been ad hoc, and the potential benefits of reuse have never been fully realized. Systematic reuse offers the greatest potential for significant gains in software development productivity and quality. The strategy for adopting a reuse technology should be based on a vision for improving the organization's way of doing business. They presented a Reuse Maturity Model proposal, describing consistence features for the incremental reuse adoption.

Postmus et al., [Aligning the economic modelling of software reuse with reuse practices, 32] believe, in contrast to current practices where software reuse is applied recursively and reusable assets are tailored trough parameterization or specialization, existing reuse economic models assumes two things. First, the cost of reusing a software asset depends on its size and second, reusable assets are developed from scratch. They provided modelling elements and an economic model that is better aligned with current practices.

Haefliger et al., [Code reuse in open source software, 33] believe, code reuse is a form of knowledge reuse in software development, which is fundamental to innovation in many fields. They found that code reuse is extensive across the sample and that open source software developers, much like developers in firms, apply tools that lower their search costs for knowledge and code, assess the quality of software components, and they have incentives to

reuse code. Open source software developers reuse code because they want to integrate functionality quickly, because they want to write preferred code,

because they operate under limited resources in terms of time and skills, and because they can mitigate development costs through code reuse.

Mohagheghi et al., [An Empirical Investigation of Software Reuse Benefits in a Large Telecom Product, 34] describe a case study on the benefits of software reuse in a large telecom product. The reused components were developed in-house and shared in a product-family approach. Quantitative data mined from company repositories are combined with other quantitative data and qualitative observations. They observed significantly lower fault-density and less modified code between successive releases of reused components.

Jansen et al., [Pragmatic and Opportunistic Reuse in Innovative Start-up Companies, 35] found that reuse of software components and services enables software vendors to quickly develop innovative applications and products. Integration of functionality cannot be performed successfully without formal component and service selection and integration procedures. It shows two products that have been developed successfully by two start-up companies, using a pragmatic approach to third-party functionality reuse and integration.

Gibson, [Software Reuse and Plagiarism: A Code of Practice, 36] found that university guidelines or policies on plagiarism are not sufficiently detailed to cope with the technical complexity of software. In his policy, he specifies the notion of software to cover all the documents that are generally built during the engineering of a software system — analysis, requirements, validation, design, verification, implementation and tests. He concluded with a simple code of practice for reuse of software based on a file-level policy, combined with emphasis on re-using only what is rigorously verified.

Kath et al., [Safety, Security, and Software Reuse: A Model- Based Approach, 37] believe, with the move to distributed, component based systems involving reuse of components and services, emergent, system-wide properties, including safety and security in particular, are becoming increasingly difficult to guarantee. Model based techniques constitute a promising approach to guarantee safety and security in systems built with reusable components. The key elements in this approach are correctness and certifiability by construction, and separation of concerns.

Arman et al., [E-learning Materials Development: Applying and Implementing Software Reuse Principles and Granularity Levels in the Small, 38] exclaimed that it has always been a challenge to identify proper e-learning materials that can be reused at a reasonable cost and effort. E-learning materials development is typically acknowledged as an expensive, complicated, and lengthy process, often producing materials that are of low quality and difficult to adapt and maintain. Here, software engineering reuse principles are applied to e-learning materials development process. These principles are then applied and implemented in a prototype that is integrated with an open-source course management system.

Dantas et al., [Software Reuse versus Stability: Evaluating Advanced Programming Techniques, 39] found that with system development becoming increasingly incremental, software reuse and stability stand out as two of the most desirable attributes of high-quality software. A key goal in contemporary software design is to simultaneously promote reuse and stability of the software modules. There are a growing number of techniques for improving modularity, ranging from aspect-oriented and feature-oriented programming to composition filters. They presented an exploratory analysis of advanced programming techniques on how it is made possible to reach a better trade off of software reuse and stability. Results revealed that a hybrid incarnation of feature-oriented and aspect-oriented programming seems to be the most promising programming technique.

Martínez-Fernández et al., [REARM: A Reuse-Based Economic Model for Software Reference Architectures, 40] aimed at ameliorating the lack of support for evaluating the economic impact of decisions with regard to software reference architectures, by presenting a pragmatic preliminary economic model to perform cost-benefit analysis on the adoption of software reference architectures as a key asset for optimizing architectural decision-making. The model is based on existing value-based metrics and economics-driven models used in other areas. A preliminary validation based on a retrospective study showed the ability of the model to support a cost-benefit analysis presented to the management of an IT consulting company. This validation involved a cost-benefit analysis related to reuse and maintenance.

3. Software Reuse Methodologies

3.1 Software Reuse Principles

An excellent set of candidate principles for software

reuse [37, 38]:

1. Build a software domain as an architecture and as a framework for the reuse activities.
2. Use a typical software development process which will promote and control reuse.
3. Reuse more than just code.
4. Practice domain engineering.
5. Integrate reuse into project and quality management and also software engineering activities.
6. Organize the enterprise to facilitate partnering to achieve reuse across product boundaries.
7. Use automation tools to support reuse.
8. Couple modern reuse theory and technology with natural, traditional organizational reuse practices for effective results.
9. Concept, Content and Context.

The principles can be characterized in terms of each reuse methodology in terms of its reusable relics their abstraction, selection, integration and specialization [14]. We have studied various principles to solve the software reuse problems and these have been mentioned in the flowing part. REBOOT strategy implements the solutions offered by the organizational and technical aspects put forward by the REBOOT methodology [18]. Another one is the classification by means of faceted classification and its deployment to production environments [11]. The next one describes the problems of indexing and retrieval for software repositories and to present a set of tools designed to support an incremental refinement of component repositories [19]. Another principle that was studied in this process is a software construction method with guidelines for using designed reusable C++ components [16]. Apart from this, an experimental software repository system that provides organization, storage, management, and access facilities for reusable software components was another stable principle [17] Another major field of study is the development of CodeFinder to design of tools to help software designers build repositories of software components and locate potentially reusable software in those repositories [13].

Software reuse is generally defined as the use of previously developed software resources from all phases of the software life cycle in new applications by various users such as programmers and systems analysts [22]. Reuse changes the process of development from the analysis, design, and implementation of customized software to the development of reusable components applied to develop a stream of applications (Basset 1997) [29]. Software reuse is a process innovation that strives to maximize the use of reusable components in building a stream of software applications within a specific domain (Prieto-Diaz 1993;Tracz 1987) [29]. Software reuse may be focused on the exploitation of internally developed systems components or may make use of the emerging component ware market, where prices are often a fraction of the costs of home-grown development. The two key benefits of software reuse are that: (1) those components that have already been tested provide higher guarantees of robustness and reliability in any future implementation; (2) component reuse should lead to faster development times and lower costs [26]. The OO approach (for which AUTOSOFT is principally designed) is recognized to facilitate software reuse [26] because it removes the limitations imposed by monolithic, tightly coupled software development. AUTOSOFT should therefore also be suitable for supporting BPR initiatives as generic objects can be (re)incorporated into code with greater ease than software written using non-OO approaches. Reuse of software entails the design and implementation of general software systems ("reusable functional collections," "generic systems"), which perform frequently used, common, and repetitive data processing tasks. Typical examples are operating systems, compilers, database management systems, and mathematical subroutine packages. [9]

It is hard to overstate the importance of using a high level language. Many Unix systems have the source code for commands, libraries, etc., on-line and accessible to all users. Accordingly, it is straightforward to find a program, read its code, and use that as a model or a starting point for a new program. Sharing assembly language code is far harder; the set of people who can read and profit from it is much smaller [4].

3.2 Software Reuse Benefits

Although software reuse has many evident benefits, it also has many inevitable problems (discussed next). The benefits are [38]:

1. Increased dependability
2. Reduced process risk
3. Better resource utilisation
4. Effective use of specialists
5. Standards compliance
6. Saves Time
7. Accelerated development
8. Improvements in productivity
9. Improvements in quality

One major benefit in terms of source code

components is the benefit of using them as abstraction specifications for reusable components; that is, it makes the cognitive distance relatively high [14]. Reuse technology when implemented through the REBOOT method can be useful to be able to retrieve not only components exactly matching the search criteria but also components that are close to a match [18]. Apart from this, implementing techniques like faceted classification offers certain features that not only contribute to the search and retrieval, but support the potential reuse's selection process and help in the development of a standard vocabulary for software attributes [11]. Besides, reuse involves more than just code; it includes organizing and encapsulating experience and set up of the mechanisms and organizational structures to support such process. The reuse community has started emphasizing that reuse is not only a technical problem but it involves several other factors [12].

There are several elements defined during the analysis, design and implementation of a simulation model, as ideas, concepts, object classes etc. that should be reused in new applications. Application of reuse methodology and practice will reduce the effort in developing new simulation models to assist the design of new systems [23].Reuse is also expected to improve the quality of software because reusable assets having gone through multiple iterations of testing and quality enhancements are likely to have fewer errors than freshly developed code. It is reasonable to consider the possibility of leveraging software reuse in attempting to increase reliability and decrease costs and effort of producing software products. Software reuse is widely believed to be the most promising technology for significantly improving software quality and productivity (Frakes, 1992). It is believed that constructing systems from existing software components should shorten development time, lessen duplication across projects, and lower development costs. One can easily see the productivity gains from not writing code (Tirso and Gregorious, 1993) [22]. In plain language, reuse is based on the principle of not reinventing the wheel (Troy, 1994) [27].

The successful practice of software reuse appears to help considerably in the teaching of certain kinds of computer science courses. For example, in a compiler construction lab course at Irvine, we introduce a compiler, called SmallGol, study it for three weeks intensively to understand in detail how it works, modify it to handle extended classes of statements and expressions, and then start from scratch to write a completely new compiler for a small portion of Ada emphasizing reuse of the parts of the SmallGol compiler. [1]

The use of C and the public accessibility of the source code for programs have led to a shared style of programming, a set of conventions, and the reuse of ideas, algorithms, and source code. The reuse of source code in this way supplements the usual object library mechanism. [4]

3.3 Software Reuse Problems

Few major problems of software reuse are [38]:

10. Increased maintenance costs
11. Tackling plagiarism
12. Lack of tool support
13. Not-invented-here syndrome
14. Creating and maintaining a component library
15. Finding, understanding and adapting reusable components

Since software is being reused, its stability after being modified is a concern for performance [39]. Tackling plagiarism is a big challenge and a model to check its level is proposed in [36]. Safety and security both are not the same but different, and should be addressed when we talk about reuse [37].

The major issue is not lack of technology but high unwillingness to address the most important issues influencing software reuse. This includes the managerial, legal, economic, cultural, and social issues. On one hand we have our technical toolbox full but on the other, we cannot use these tools effectively because a proper infrastructure is absent [12]. One major problem in software reuse is potentially organizing the collections of reusable components being implemented for effective search and retrieval of reusable software components [12]. Another issue is in terms of "generality of applicability versus payoff." Certain technologies are very general and can be implemented in a wide range of application domains. From the producer's perspective, this usually implies a much lower payoff for each individually-reused module than the systems that are focus on one or two application domains [20].

For the improvement of efficiency of software development, domain engineering has not proved to be very effective due to possible reasons like lack of customizability, misguided scoping of application area etc. [24]. The issue with regard to user classes is that great difficulty has been experienced in extending reuse beyond a single developer reusing his or her own software resources [22]. The next set of issues concerns what can be feasibly and economically reused. Software resources can be classified according to entity type, level of abstraction (or stage in the development life cycle in which they are produced), and application type .The final set of issues involves the type of software

task, which may vary from maintaining existing systems to developing new software[22]. The issue of retrieving components from software libraries has captured the attention of the software reuse community [Burton et al. 1987; Devanbu et al. 1991; Frakes and Gandel 1990; Frakes and Nejmeh 1987; Frakes and Pole 1994; Maarek et al. 1991; Prieto-D´ıaz and Freeman 1987; Sommerville and Wood 1986] [2].There are a few barriers in reuse adoption. While reuse promises a quick time to market and a reduction in cost, it requires an investment in both time and resources to deliver these benefits [27]. Several practitioners view resistance as a consequence of ‘‘weak collaborative praxes’’ embedded within the hierarchical divisional structure of the firm inhibiting inter-team collaboration [21].

A chronic problem of the programming industry has been the lack of a standard data interchange format, to facilitate both sharing data among applications and to facilitate widespread program reusability. [2]

The only negative aspect of being able to read and understand programs is that people do read and understand, and then feel free to modify. The result is sometimes an improvement in an old program, but is all too often merely a difference, leading to a proliferation of variants. Few programs are unscathed by gratuitous tinkering. Far more serious, however, is that there are even within Bell Labs three or four major versions of the operating system in widespread use, sufficiently different that some programs need conditional code to compile and run properly on all of them. The increasing number of microprocessor Unix systems threatens to make the situation worse. [4]

3.4 Software Reuse Solutions

Software reuse has several domain specific problems which are difficult to identify in the first place and then resolve.

For software reuse, stability plays an important role when it comes to performance for this, a model which tests Modularity vs. Stability vs. Reuse was proposed. Plagiarism is an commonly discussed and to resolve copyright issues and conflicts it should be kept in check while software reuse (specifically code reuse). Some measures which can be taken are proposed in the paper [36]. A reuse with a database specific approach is explained in the paper [38] where granularity is taken into account. The paper [31] suggests a Reuse Maturity Model proposal, describing consistence features for the incremental reuse adoption. Start-up Companies have started pragmatic and opportunistic reuse to quickly develop innovative applications and products [35]. If reuse is difficult to understand and implement, the 3C’s model in the paper [37] can help resolve it along with safety and security issues.

Software reuse has several domain specific problems which are difficult to identify in the first place and then resolve.

For software reuse, stability plays an important role when it comes to performance for this, a model which tests Modularity vs. Stability vs. Reuse was proposed. Plagiarism is an commonly discussed and to resolve copyright issues and conflicts it should be kept in check while software reuse (specifically code reuse). Some measures which can be taken are proposed in the paper [36]. A reuse with a database specific approach is explained in the paper [38] where granularity is taken into account. The paper [31] suggests a Reuse Maturity Model proposal, describing consistence features for the incremental reuse adoption. Start-up Companies have started pragmatic and opportunistic reuse to quickly develop innovative applications and products [35]. If reuse is difficult to understand and implement, the 3C’s model in the paper [37] can help resolve it along with safety and security issues.

For a software reuse method to be effective it should be reducing the cognitive distance between the initial concept of a system and its final executable implementation and the software abstraction model tries to work its way around this [14]. The REBOOT methodology tries to overcome the difficulties involved with software reuse by paying attention both to organizational as well as technical aspects at the same time putting forward a reuse methodology that could be integrated with the present methods generally going for technology consolidation instead of innovation [18]. The system states that a majority of the research done has constantly focused on looking for literature references, where certain predefined relationships between words and concepts or the regular structure of language can be used to effectively retrieve relevant documents. It tries to change the fact that little attention has been given to the representation of non-text data such as software components, which present some special problems not found in text-based retrieval systems [13]. Another simple method implements the predictive model which uses the reuse data, and the observed resource and quality metrics are measured and compared with the estimated values [20] thus helping to narrow down the type of methodology to be used in similar cases in the future.

Many software reuse solution models have been proposed. PULSE (Product Line Software Engineering)

methodology has been proposed for the purpose of enabling the conception and deployment of software product lines [24].The reuse of FMS components for new modelling applications in manufacturing sector has been demonstrated using the SALMS 2 tool. The test of the simulation model is done by running the SSS software .Frameworks can

be used to organize the existing literature on software reuse, to evaluate trends, and to point to areas needing further study [23]. Freeman developed a framework based on three dimensions: the artefact being reused, the manner in which it is reused, and the factors needed to enable successful reuse [22]. Another paper motivates, presents, demonstrates in use, and evaluates a methodology for conducting design science (DS) research in information systems (IS). The design of this conceptual process will seek to meet three objectives: it will (1) provide a nominal process for the conduct of DS research, (2) build upon prior literature about DS in IS and reference disciplines, and (3) provide researchers with a mental model or template for a structure for research outputs [30]. A C/C++based methodology enables hardware-software co-design and gives designers the ability to perform hardware-software co-verification and performance estimation at very early stages of design. In this paper we show how hardware-software co-verification is performed in a C/C++-based flow. In particular, we will use the SYSTEMC (formerly known as SCENIC) environment [25] throughout the design flow. The architecture/component technology has been developing promptly among numerous software reuse technologies in recent years [28] like Open CNC and the software architecture/component reuse technology, the architecture of open CNC system and the integrated platform of developing CNC system.

Under the general heading of "reusability" are five important subtopics which are the subjects of this report: Reusable data, Reusable Architecture, Reusable Designs, Reusable Programs and Common Systems, and Reusable Modules. The underlying concept of data reusability is that storage will become so inexpensive that information about data, which by volume may take more space than the actual data records, can utilized to identify the attributes of the data. This will allow truly universal reusability of data, because the expanded records labels will allow applications programs to utilize data without needing prior agreement between the application programmer and the data administrator in terms of formats and content, assuming of course that the data labels are in a standard form. [2]

One advantage of Stroustrup's approach, as compared to Ada or CLU, is that it builds on a successful existing language instead of defining an entirely new one. This retains the assets of the base language, including its efficiency, portability, and established user population. Users can work into the class system gradually, beginning with its better checking of vanilla C, before learning the new capabilities. [4]

4. Conclusion

To conclude our survey, we return to the problem of how to practice software reuse. This problem won't have any one common answer because of the fact that software reuse has several factors which are context and domain specific. So depending upon in which specific domain of software engineering we wish to practice software reuse, some norms and procedures have to be followed which we have discussed and addressed in our survey. It is really not possible to find out a general software reuse mechanism that'll suit all purposes and needs. So we have taken several domain specific reuse mechanisms and analysed each one of them. Finally, the results of the analysis have been presented. The major conclusions are that, tools should be used for software reuse to maintain quality of product, plagiarism should be kept in check and domain specific software reuse strategies should be adopted for best results.

References

[1] Standish, T. A. (1984). An essay on software reuse. IEEE Transactions on Software Engineering, (5), 494-497.

[2] Jones, T. C. (1984). Reusability in Programming: A Survey of the State of the Art. IEEE Transactions on Software Engineering, (5), 488-494.

[3] Neighbors, J. M. (1984). The Draco approach to constructing software from reusable components. IEEE Transactions on Software Engineering, (5), 564-574.

[4] Kernighan, B. W. (1984). The Unix system and

software reusability. IEEE Transactions on Software Engineering, (5), 513-518.

[5] Jones, B., Litvintchouk, S., Mungle, J., Krasner, H., Mellby, J., & Willman, H. (1985). Issues in software reusability. ACM SIGAda Ada Letters, 4(5), 97-99.

[6] McCain, R. (1985, October). Reusable software component construction-A product- oriented paradigm. In 5th Computers in Aerospace Conference (p. 5068).

[7] Anderson, C., & McNicholl, D. (1985, October). Reusable software-A mission critical case study. In 5th Computers in Aerospace Conference (p. 5069).

[8] Honiden, S. H. I. N. I. C. H. I., Sueda, N., Hoshi, A., Uchihira, N. A. O. S. H. I., & Mikame, K. A. Z. U. O. (1986). Software prototyping with reusable components. Journal of Information Processing, 9(3), 123-129.

[9] Polster, F. J. (1986). Reuse of software through generation of partial systems. IEEE transactions on software engineering, (3), 402-416.

[10] Frakes, W. B., & Nejmeh, B. A. (1986, September). Software reuse through information retrieval. In ACM SIGIR Forum (Vol. 21, No. 1-2, pp. 30-36). ACM.

[11] Prieto-Diaz, R. (1991). Implementing faceted classification for software reuse. *Communications of the ACM*, *34*(5), 88-97.

[12] Prieto-Díaz, R. (1991). Making software reuse work: an implementation model. *ACM SIGSOFT Software Engineering Notes*, *16*(3), 61-68.

[13] Henninger, S. (1995). Information access tools for software reuse. *Journal of Systems and Software*, *30*(3), 231-247.

[14] Krueger, C. W. (1992). Software reuse. *ACM Computing Surveys (CSUR)*, *24*(2), 131-183.

[15] Isoda, S. (1992, June). Experience report on software reuse project: its structure, activities, and statistical results. In *Proceedings of the 14th international conference on Software engineering* (pp. 320-326). ACM.

[16] Chen, D. J., & Lee, P. J. (1993). On the study of software reuse using reusable C++ components. *Journal of Systems and Software*, *20*(1), 19-36.

[17] Constantopoulos, P., Jarke, M., Mylopoulos, J., & Vassiliou, Y. (1995). The software information base: A server for reuse. *The VLDB Journal—The International Journal on Very Large Data Bases*, *4*(1), 1-43.

[18] Sindre, G., Conradi, R., & Karlsson, E. A. (1995). The REBOOT approach to software reuse. *Journal of Systems and Software*, *30*(3), 201-212.

[19] Henninger, S. (1997). An evolutionary approach to constructing effective software reuses repositories. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, *6*(2), 111-140.

[20] Leach, R. J. (1997). *Software Reuse: methods, models, and costs*. New York: McGraw- Hill.

[21] Rine, D. C., & Sonnemann, R. M. (1998). Investments in reusable software. A study of software reuse investment success factors. *Journal of systems and software*, *41*(1), 17- 32.

[22] Kim, Y., & Stohr, E. A. (1998). Software reuse: survey and research directions. *Journal of Management Information Systems*, *14*(4), 113-147.

[23] Kovacs, G. L., Kopácsi, S., Nacsa, J., Haidegger, G., & Groumpos, P. (1999). Application of software reuse and object-oriented methodologies for the modelling and control of manufacturing systems. *Computers in Industry*, *39*(3), 177-189.

[24] Bayer, J., Flege, O., Knauber, P., Laqua, R., Muthig, D., Schmid, K., ... & DeBaud, J. M. (1999, May). PuLSE: a methodology to develop software product lines. In *Proceedings of the 1999 symposium on Software reusability* (pp. 122-131). ACM.

[25] Séméria, L., & Ghosh, A. (2000, January). Methodology for hardware/software co- verification in C/C++ (short paper). In *Proceedings of the 2000 Asia and South Pacific Design Automation Conference* (pp. 405-408). ACM.

[26] Gibb, F., McCartan, C., O'Donnell, R., Sweeney, N., & Leon, R. (2000). The integration of information retrieval techniques within a software reuse environment. *Journal of Information Science*, *26*(4), 211-226.

[27] Sherif, K., & Vinze, A. (2003). Barriers to adoption of software reuse: a qualitative study.*Information & Management*, *41*(2), 159-175.

[28] Bin, L., Yun-fei, Z., & Xiao-qi, T. (2004). A research on open CNC system based on architecture/component software reuse technology. *Computers in Industry*, *55*(1), 73- 85.

[29] Sherif, K., Zmud, R. W., & Browne, G. J. (2006). Managing peer-to-peer conflicts in disruptive information technology innovations: the case of software reuse. *MIS quarterly*, 339-356.

[30] Peffers, K., Tuunanen, T., Rothenberger, M. A., & Chatterjee, S. (2007). A design science research methodology for information systems research. *Journal of management information systems*, *24*(3), 45-77.

[31] Garcia, V. C., Lucrédio, D., Alvaro, A., De Almeida, E. S., de Mattos Fortes, R. P., & de Lemos Meira, S. R. (2007). Towards a Maturity Model for a Reuse Incremental Adoption. In SBCARS (pp. 61-74).

[32] Postmus, D., & Meijler, T. D. (2008). Aligning the economic modeling of software reuse with reuse practices. Information and Software Technology, 50(7), 753-762.

[33] Haefliger, S., Von Krogh, G., & Spaeth, S. (2008). Code reuse in open source software. Management Science, 54(1), 180-193.

[34] Mohagheghi, P., & Conradi, R. (2008). An empirical investigation of software reuse benefits in a large telecom product. ACM Transactions on Software Engineering and Methodology (TOSEM), 17(3), 13.

[35] Jansen, S., Brinkkemper, S., Hunink, I., & Demir, C. (2008). Pragmatic and opportunistic reuse in innovative start-up companies. IEEE software, 25(6).

[36] Gibson, J. P. (2009, July). Software reuse and plagiarism: a code of practice. In *ACM SIGCSE Bulletin* (Vol. 41, No. 3, pp. 55-59). ACM.

[37] Kath, O., Schreiner, R., & Favaro, J. (2009, September). Safety, security, and software reuse: a model-based approach. In *Proceedings of the fourth international workshop in software reuse and safety*.

[38] Arman, N., & Hebron, P. (2010, June). E-learning materials development: Applying and implementing software reuse principles and granularity levels in the small. In *Proceedings of the International Conference on E-Learning, E-Business, Enterprise Information Systems, & E-Government*.

[39] Dantas, F., & Garcia, A. (2010, September). Software reuse versus stability: Evaluating advanced programming techniques. In *Software Engineering (SBES), 2010 Brazilian Symposium on* (pp. 40-49). IEEE.

[40] Martínez-Fernández, S., Ayala, C. P., Franch, X., & Marques, H. M. (2013, June). REARM: A reuse-based economic model for software reference architectures. In *International Conference on Software Reuse* (pp. 97-112). Springer Berlin Heidelberg.