A Study of Semi-Automated Program Construction

H. Dayani-Fard[1] J. I. Glasgow[2] D. A. Lamb[3]

June 23, 1998
External Technical Report
ISSN-0836-0227-
1997-416

Department of Computing and Information Science
Queen's University
Kingston, Ontario, Canada K7L 3N6

Document prepared June 23, 1998

[1]IBM Centre for Advanced Studies
[2]Department of Computing and Information Science, Queen's University
[3]Department of Computing and Information Science, Queen's University

Contents

List of Figures

1 Introduction

The term "Software Engineering" was supposedly first introduced at the 1968 NATO software engineering conference [31], the theme of which was the "software crisis." From that date, computer applications have grown in number, size, and complexity but the "crisis" still remains. The modern definition of the software crisis relates to the problem of building and maintaining large, complex, and reliable software systems in a controlled and cost effective way [24, 32]. Thus far, the growth of software needs has been addressed in part by the increase in the number of trained professionals. However, this trend cannot continue [25].

A proposed alternative solution to the software crisis, since the early days of computers, has been the use of automatic programming techniques. In the 1950's, a Fortran compiler was considered by many to be an automatic programming system [4]: high-level programs could be translated automatically into machine language. Today, even very high-level languages can hardly qualify as automatic programming systems. Automatic programming is a moving target, which reflects the increasing expectation of automation [38].

The main idea behind automatic programming is to translate a "specification" of a problem into an executable program with little or no intervention on the user's part. The false assumptions in this characterization of automatic programming are: 1) one's specification can be another one's implementation, 2) specifications are often incomplete, and 3) for every specification there are combinatorially many implementations [27]. As a result, an automatic programming system that takes as an input a (general) specification and results in an implementation is far from reality.

An alternative approach to improving software construction involves a hybrid technique: a combination of AI and traditional software reuse techniques. The main motivation behind this hybrid approach is the nature of the software itself. Software has been compared to hardware IC's and the process of software construction as the composition of the software components [29]. This analogy was considered *de facto* by most researchers until it was suggested (for example, see Software Reusability [8]) that unlike hardware components software does not wear or tear: there is no physical representation of software and it is difficult to construct abstractions for software components. Our abstractions are based on how we want to perceive software components; there is no physical object that we can compare our abstractions against. Furthermore, unlike most engineering products, software can change over its life time. Soft-

ware resembles human thoughts more than any physical artifacts [8]. Software is a solution to a problem which when coded in some programming language communicates our intentions with the computer. In other words, we can view software as a static representation of our knowledge of a solution to a particular problem.

This view is consistent with the results of research in component libraries [4]. Component libraries do not store the history of design: the decisions made and the rationales behind them are lost. In effect, the knowledge gained through the design process is absent in the final software. The evolution of software attests to the importance of this history. In general, software systems were designed with the assumption that their life time would be short, but many have out-lasted their life expectancy. Modifications, changes of requirements, and upgrades have caused the design history –if it was documented– to be outdated and in most cases of little or no use.

In recent years, the issue of capturing software engineering knowledge has received fair attention. The problems of software evolution, program understanding, and reverse engineering have been identified as the direct consequences of the lost knowledge of the original development process; this knowledge is in the minds of the people who may no longer be available to help in the evolution of the software. Domain knowledge has also received much attention. In fact, it has been suggested that software engineering, due to the diverse computational needs, is no longer a homogeneous field: the knowledge of the domain of application plays a crucial role in the success of software engineering [20, 21].

For the reasons mentioned above, a new approach to software engineering has evolved: *knowledge-based software engineering* [27]. The idea behind knowledge-based software engineering is that programmers reuse their knowledge when designing new software systems. Knowledge-based software takes advantage of AI techniques for knowledge representation to capture the design decisions, problem solving knowledge, domain knowledge, and other important aspects of the history of software systems. Simply stated, knowledge-based software engineering can be viewed as the application of expert systems to the domain of software engineering [27].

There have been many efforts invested in knowledge-based software engineering (e.g. KIDS [45], GLITTER [14], and DRACO [32]). There are special issues of journals and conferences that are dedicated to this topic. However, at the time of writing this paper, there has not been any comparative study

of the applications of AI techniques to software engineering with emphasis on semi-automated program synthesis.

The AI techniques used in software engineering have one common aspect: they try to capture and reuse the knowledge needed to design and implement software. We can classify these approaches based on their methods of knowledge representation. In the rule-based approach, knowledge about the domain and/or software design is formally expressed using production rules. This approach is similar to transformational systems, where the transformations are rules and their applications can be determined using the captured knowledge. Examples include DRACO [33], TI [4], and GLITTER [14].

An alternative approach uses case-based reasoning. The fundamental idea of case-based reasoning is that problem solvers use their previous experiences – called cases– to solve new problems [22]. Cases are semi-formal representations of individual instances of problem solving knowledge which can be reused to solve new problems. The main tasks involved in case-based reasoning are abstraction, selection, adaptation, evaluation, and storage [22]. These tasks are similar to Krueger's list of the fundamental issues in software reuse [24].

The ideas behind knowledge-based software engineering seem reasonable. However, despite all the efforts spent over the past decade, there are still not many functioning systems that can be used by industry practitioners. The main motivation of this study is to compare a sample of different approaches to semi-automated program synthesis and identify strengths and shortcomings of each approach. Further, we can draw some conclusions based on this study as what the future directions should be.

This paper presents a study of a sample of research efforts in semi-automatic program construction. These efforts all focus, to varying degrees, on recording and employing some forms of experts' knowledge and the knowledge of the application domain. Further, they provide mechanisms to allow for some degree of automation in program construction. The results of this study identify with the realities of automatic programming as outlined by Rich and Waters [38]:

1. system specifications are rarely complete; an iterative process is required to identify the inconsistencies and incompleteness in the specification,

2. knowledge of the application domain is needed to allow for effective communication between the user and the system,

3. automatic programming systems must allow for the user's interaction and provide assistance in the decision making process, as opposed to full

automation based on a predetermined policy that excludes users during all or some of the development stages, and

4. the history of program construction, the decisions made during the design, and the rationales behind them must be recorded in such ways that they can be reused in the future.

The systems selected for this study are representatives of different approaches that have persisted over time. The study is not exhaustive and we focus on the chronological progress of automatic programming concepts rather than the developed systems. The Transformational Implementation [4] and the Programmer's Apprentice [47], discussed in Section 2.1, have been exemplars of the semi-automatic programming paradigm, while GLITTER [48], PADDLE [14], and Draco [32] have tried to improve upon the lessons learned from these approaches. In Section 2.2, we study some of the newer research efforts; though they have not shown their persistence, they offer a different approach to tackling semi-automated program construction.

This survey is organized as follows. Section 2 presents a study of selected semi-automatic programming systems. This section is divided into two subsections: rule-based systems and case-based systems. This division is not accurate according to the traditional AI definitions of rule-based and case-based systems. We categorize each approach based on its resemblance to one of the two categories: if a system records the expert's knowledge in a form similar to production rules, we categorize it as a rule-based system; if a system records the expert's experiences, we categorize the system as a case-based system.

Our study concludes in Section 3 by providing a comparison of different approaches based on 1) their knowledge acquisition ability, 2) their initial cost, 3) the level of support they provide in decision making and software evolution, and 4) their overall practicality. Section 4 provides some remarks on the future directions of semi-automatic programming systems.

2 Semi-Automated Programming Systems

There have been many studies done on the applications of AI techniques to software engineering (for example, see Lowry [26]). More and more, researchers and practitioners try to find ways to employ AI to achieve the goals of higher reliability and shorter design time in software engineering in general and software reuse in particular. This trend stems from the thought that in order

to solve difficult problems using computers, one will generally have to use a great deal of domain specific knowledge rather than a few general principles [9]. To be able to use this knowledge automatically, or semi-automatically, it must be represented in some manner inside the computer. Some of the existing knowledge representations used in main stream AI can be readily used in the domain of software engineering; others must be altered to better represent software engineering knowledge. Furthermore, new knowledge representations need to be devised to capture shortcomings of other methods mainly with respect to non-functional requirements (for example, see Telos [23]).

In this section, I describe some AI approaches to software reuse. This study is by no means comprehensive. Instead, I have included those approaches that focus on the problem of the lost history of software. As described previously in Section 1, during the design of a software system, design decisions, the rationales behind them, and other motivations, are poorly captured or lost altogether. For this reason, in this study I focus on semi-automated program synthesis systems which attempt to remedy this problem.

The systems studied here are divided into two sections. First, I look at systems that use formal languages to capture and represent knowledge about a software artifact (specification, design, etc). Then, I look at systems that use a form of AI reasoning, case-based reasoning, which is less formal and relies heavily on the notion of *similarity*. For the sake of this study, I loosely categorize the former under *rule-based* systems. These systems, more or less, rely on the rule-based knowledge representation schemes. They represent software engineering knowledge in the form of rules that can be manipulated to provide a semi-automated program synthesis system. The latter systems, I categorize under *case-based reasoning* systems. Several examples from both categories are presented and compared.

2.1 Rule-Based Systems

This section presents a study of program synthesis approaches that are loosely categorized under rule-based systems. The justification for this categorization stems from the resemblance between these approaches and traditional expert systems. Expert systems generally capture the domain knowledge in the form of production rules and provide inference engines for reasoning and manipulating these rules. All approaches studied in this section, to some extent, follow this paradigm. They generally use a set of transformation rules and some form

of automated reasoning to provide suggestions for selecting a rule, apply rules, consistency check, constraints satisfaction, etc.

The approaches studied in this section can be classified as program transformation systems. In general, program transformation systems enable the designer to begin with a high-level specification and, using the transformation rules, refine the specification to an implementation. Transformation rules are mappings from one program to another:

$$P \times T \rightarrow P'$$

where P is a program, T is a transformation rule, and P' is the resulting program. The most important semantic relation of a transformation rule is program *equivalence*, where there is an inverse rule for transforming a program back to its original form (i.e., a one-to-one relation). The next important relation is *weak equivalence* where undefined situations are ignored; this enables a designer to ignore certain situations such as error condition.

Transformation rules vary in form, some are syntactic rules which relate language constructs, for example a loop construction rule can be stated as:

$$L : \text{if } \ b \text{ then } S; \text{ GOTO } L = \ \text{while } \ b \text{ do } \ S$$

Others are domain rules which express domain knowledge, for example:

$$\text{pop}(\text{push}(s, x)) \leftrightarrow s$$

for an unbounded stack s.

The advantage of using transformation systems stems from the formalization of incremental changes to a program. In another words, all changes to the initial specification can be recorded as the design history and replayed or backtracked at a later time.

The early transformation systems were manual; the designer had to select a transformation and apply it. More ambitious systems later on attempted to fully automate the transformational process. As an example consider PSI [10], which starts with an abstract algorithm and automatically refines it to an executable program. Due to the narrow domain of application for a fully automated system on one end and the lack of any form of automation in fully manual systems, semi-automated systems were introduced. The motivation behind semi-automation was that the designers need decision support rather than full automation. The newer systems attempted to include a knowledge-base in

their systems to provide support in the decision making process. Further, the commercially available systems, such as KIDS [44] from Kestrel Institute, were built on top of available knowledge-bases.

The next section presents three semi-automated approaches. First, the Transformational Implementation paradigm is studied, which is one of the earliest attempts at semi-automated program synthesis. Next, we study the Programmer's Apprentice project, which deviates from mainstream transformational systems, and lastly we present the Draco system, which focuses on domain analysis to capture knowledge about software systems.

2.1.1 Transformational Implementation

The program transformation paradigm has been studied as a means of program synthesis since the early 1970's (e.g. SAFE [2], TI [4], PADDLE [48]). The basic idea behind this paradigm is to apply a set of transformations to a formal specification of a system to produce an efficient implementation. The process of transformation application can be manual, semi-automated, or fully automated. In all cases, systems are interactive and need some degree of user input.

Transformational systems, mostly, have a predefined collection of transformation rules called a *catalog*. A catalog is a hierarchically structured collection of transformation rules relevant to a particular aspect of the development process [34]. Catalogs may contain rules about solution strategies such as binary search, optimization rules such as recursion removal, or domain knowledge such as set operations.

The goal of the transformational paradigm is to assist in producing reliable and efficient implementations. Further, some systems based on this paradigm record decisions behind transformations applied, while others provide replay mechanisms which enable the designer to go back to an arbitrary point in time and reconstruct the derivation of the system. Such facilities simplify the maintenance task; instead of maintaining the code, modifications are performed on the specification of the system and the derivation is repeated. However, if the design history is lost or recorded partially, it becomes extremely difficult to perform maintenance tasks.

One of the early attempts at using the transformational paradigm for program synthesis was made by Balzer [2]. This work focused on the project SAFE [6]. The motivation behind this project was to acquire and validate a specification using an operational specification language, which could then

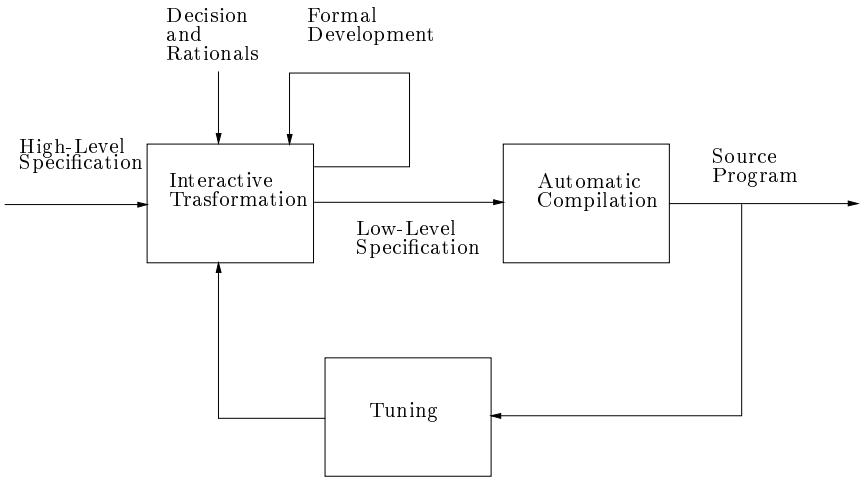

Figure 1: Automation-Based Software Paradigm (Adapted from Balzer 1985)

be translated into an efficient implementation. The latter part of the project, program synthesis, led to the *Transformational implementation* project (TI) [4], whose main goals were:

- to provide a means for acquiring and validating a specification,
- to record decisions employed during the design process;
- to transform a formal specification of a system into an efficient implementation.

The ideas of TI were based on the *automation-based software paradigm* [5], which influenced future projects in this area. According to this paradigm, a high-level specification can be converted to a low-level specification through an interactive transformation; the low-level specification can be automatically compiled into an efficient implementation (see Fig. 1). This paradigm includes a mechanism for recording decisions and rationales behind the selection of transformation rules as well as a tuning mechanism for improving the efficiency of the resulting implementation. According to this paradigm, developing an implementation from a formal specification is viewed as a continuous process of applying transformation rules, either to replace specification constructs or to simplify algorithmic constructs.

A side benefit of the TI project was the development of a high-level specification language called GIST [3]. GIST was developed to provide the flexibility and ease of expression needed for describing acceptable system behaviors. Furthermore, GIST is a *wide spectrum* language: it is not only a specification language, but also an implementation language for describing efficient programs. The only restriction on low-level specification is that it must be written in a subset of GIST which is automatically translatable into an existing programming language.

The main task of the programmer in TI is the selection of appropriate transformation from a pre-existing catalog. If an appropriate transformation does not exist in the catalog, the programmer may either extend the catalog or modify the program directly through an interactive editor. In either case, the responsibility for the correctness of a transformation rests with the programmer.

A key contribution of this automation-based paradigm was the notion of evolution [4]. Balzer suggested that large, complex systems cannot be predesigned; they must grow and evolve based on the feedback from the use of the system. In order to facilitate evolution, a replay mechanism must be provided to allow derivation of the system to be repeated. To do so, the history of the derivation must be recorded so that they can be applied again.

The PADDLE system [48] intended to provide what was absent in the TI project, namely the development history. The main motivation behind PADDLE was that a program development is a formal document explaining the implementation of a specification. This document, in principle, could be used by subsequent maintainers.

The development language, PADDLE, used in this project provided facilities to emphasize the structure of the program development: goals and subgoals are organized according to strategies (e.g. sequential composition, refinement subordinates, conditional) and ways of achieving them (e.g. do X by doing Y and Z). This structure enables the designers to use a larger catalog of rules; they are classified according to their strategies.

The program development process in PADDLE can be divided into five steps [48]:

1. Focus on a program fragment

2. Find an appropriate implementation strategy

3. Satisfy the conditions of the chosen strategy. (At this step, the programmer can use the interactive editor to modify the program so that the conditions of the strategy are satisfied.)

4. Simplify the resulting program.

This development process is based on the observation that a formal structure representing the transformation of a specification can partially be replayed automatically [12]. However, having only the explanations is not sufficient to enable the replay of the development process accurately [48]. The PADDLE system does not record the rationales, motivations, and assumptions behind design decisions.

The PADDLE project offers improvements in transformational implementation; the system uses a large catalog of rules organized by strategies and records the development history as a sequence of transformation applications. Its main shortcoming is its inability to record design rationales. This shortcoming was one of the motivations behind the GLITTER system.

The GLITTER [14] system was designed to improve on the shortcomings of the TI and PADDLE systems. The key goals behind this system are:

1. To provide some degree of automation for the transformation implementation.

2. To provide an interactive system to include the user in the process of transformation.

3. To record goals, subgoals, and strategies as well as decisions and rationales behind them, to provide a more complete history of the development process.

Fikas observed that the full automation of solutions to realistic problems is still unachievable [14]. Also, lack of automation in the transformational implementation can be time consuming and tedious. Hence, by including the user in a semi-automated environment, the system can achieve better results.

The GLITTER system has three types of catalogs: transformation rules, methods, and selection rules. A transformational development in GLITTER starts with some design goal expressed in the GIST language. The system, in turn, either asks the user for details or checks its *method catalog*. A method in GLITTER is a frame that has a slot for the *goal*, a slot for the *filter*, and a slot for an *action*. The goal slot is filled with the goal to be achieved, the

filter slot with predicates that will check for situations when the method is not applicable, and the action slot with one or more operations for achieving the goal. The posting of a goal causes all methods to check their goals slot. All matches are (automatically) collected together in a candidate set. If there is more than one applicable method in the candidate set, GLITTER uses the selection rules catalog to decide which method to choose.

The GLITTER system attempts to automate more portions of transformation selection and application, and leaves smaller portions to the user. For example, when the candidate set has more than one method, the system may ask the user to supply a truth value to a clause in the selection rule. The questions that GLITTER asks the user can be very difficult. However, these decisions are common to most transformational systems, with a crucial difference that these decisions are made explicit in GLITTER, whereas in TI, PADDLE, and other systems they remain implicit. GLITTER attempts to formalize goals, strategies, decisions, rationales, and all the information it is able to capture during the transformation process. This information is stored declaratively to enable the GLITTER system to behave like an expert. In other words, GLITTER can reuse (replay) its experiences partially in the development of a system.

To summarize, the TI system introduced a new paradigm in software development based on reusing transformation rules. The main goals of transformational implementation are:

- to improve system reliability
- to reduce development time
- to provide automation in the development process, and
- to allow reuse of specifications and problem solving strategies

The PADDLE and GLITTER systems, which are direct successors of TI, have solved some of the shortcomings of TI. However, there remain several issues that are common to most transformational systems. The first issue is the extension of transformation rules: none of the systems studied here provide adequate support in generating new rules. When adding a new rule to the catalog, the responsibility for ensuring that the newly added rule preserves correctness remains with the user.

In terms of automation, GLITTER provides a better balance between user interaction and system automation than PADDLE. Furthermore, GLITTER

attempts to capture its experiences, in a similar manner to that of expert systems, to reduce its reliance on the user. However, to further improve the behavior of the system, domain knowledge must be introduced into the system. This problem is partially solved by the DRACO system [18] which will be described in Section 2.1.3.

2.1.2 Programmer's Apperentice

The *Programmer's Apprentice* project (PA) started in the mid 1970's with the goal of providing intelligent assistance for software engineering tasks [47, 39, 40]. The idea behind the intelligent assistance was the IBM's Chief Programmer's Team approach [7]. In the chief programmer's team approach, a chief programmer is supported by a group of junior programmers (apprentices). The chief programmer makes the difficult decisions and performs the high-level design, while the junior programmers perform the low-level design. Further, the chief programmer could, at any time, perform the low-level design and implementation when need be.

The interaction between the programmer's apprentice and a software engineer is modeled after the interaction between a chief programmer with a human assistant. The programmer's apprentice plays an *active* role in the development process by cooperating on or taking over aspects of the software engineering tasks. However, the software engineer is always allowed to use the underlying environment directly without interaction with the programmer's apprentice. In short, a software engineer makes the hard decisions about what should be done, while the apprentice takes over much of the mundane programming tasks.

Figure 2 shows the division of labor in the programmer's apprentice project [47]. The key issue in the cooperation of a software engineer and the apprentice is their shared knowledge. This is the knowledge about the system being worked on (e.g. requirements, design) and a large shared vocabulary so that the software engineer does not have to describe everything from first principles (e.g. design ideas, implementation techniques).

To formalize the shared knowledge the programmer apprentice uses the *plan calculus* [40]. Plan calculus is a programming language independent formalization that combines flowcharts, data abstraction, and predicate calculus. The goals of plan calculus were to be expressive, provide simple facilities for combining *cliches*, and allow machine manipulatability of cliches. A cliche is a common combination of elements with familiar names ranging from low-

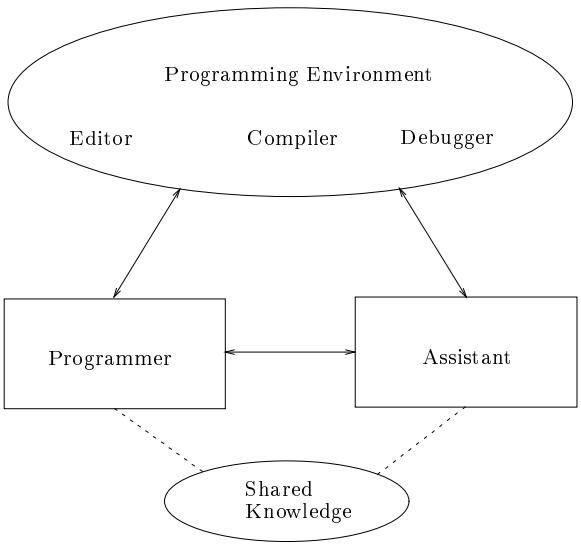

Figure 2: Division of Labor in Programmer's Apprentice (Adapted from Waters 1985)

level implementation ideas through design ideas and high-level specification concepts. Examples of cliches are device drivers, information systems, and successive approximation. Cliches are theoretical concepts; to apply these concepts, cliches are represented as *plans* [40]. A plan contains three kinds of information:

1. *plan diagrams*, which contain information about the algorithmic aspects of a plan such as data and control flow. Plan diagrams are represented using flowcharts.

2. *logical annotations*, which represent non-algorithmic aspects of a plan. These are in forms of pre- and post-condition annotations of flowcharts.

3. *overlays*, which capture the transformational aspects of a plan. Overlays are mappings between two plans. They differ from transformations in that they are bidirectional.

The underlying reasoning mechanism for plans is a multi-layer hybrid reasoning system called CAKE [37]. CAKE has several layers for reasoning about different aspects of plans (see Fig. 3). The bottom three layers of

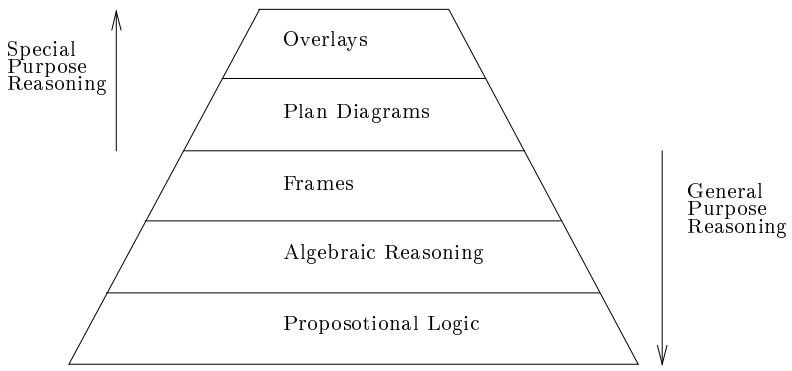

Figure 3: Layers of CAKE Reasoning System (Adapted from Rich and Waters)

CAKE (general-purpose reasoning) provide automatic one-step simple deduction, record dependencies which enables CAKE to explain its actions, detect contradictions, and verify algebraic properties and inheritance. However, the propositional logic layer of CAKE is relatively weak, which makes the practicality of CAKE to be dependent on whether most of the information in a plan is represented diagrammatically as opposed to logically.

The programmer's apprentice project focused on different aspects of software engineering tasks: requirement acquisition, design, program understanding, and program synthesis. However, this paper focuses only on its programming synthesis aspects. The demonstration system for program synthesis is a knowledge-based system which uses only a subset of plan calculus [39]. This prototype, *knowledge-based editor in Emacs* (KBEmacs), is an extension of the standard Emacs editor.

The main goal of KBEmacs was to assist in constructing programs rapidly and reliably by combining cliches. The heart of KBEmacs is a library of programming cliches or algorithmic components. The cliches are used as the shared knowledge between the programmer and KBEmacs, which can be extended by the programmer by defining new cliches. An underlying assumption in KBEmacs shared knowledge is that the programmer is at least aware of basic features of the various cliches. This assumption is reasonable only when cliches are related to a particular domain.

KBEmacs maintains two representations of the program being developed: a program text and a plan. The programmer, at any time, can modify either representation. To modify the program text, the programmer can use the

standard Emacs editor module, and to modify the plan, the user can use the knowledge-based editor module which supports several commands for instantiating and combining cliches.

When executing any knowledge-based command, KBEmacs first analyzes the effects of any editing that the programmer has done. If the plan is modified, the *coder* module of KBEmacs creates a new program text from the plan. If the program text is modified, the *analyzer* module of KBEmacs creates a new plan by analyzing the data and control flow of the program text [40]. The analyzer module is similar to the front end of an optimizing compiler. The coder module, on the other hand, is fairly complex. This complexity stems from the need to create aesthetic code. KBEmacs has comment generation capabilities, which provide significant high-level information that are not explicit in the program code.

In summary, KBEmacs as a prototype demonstrated the key ideas behind the programmer's apprentice project. First, the programmer is provided with the freedom to work on either the program text or the plan. Second, the assistant automatically modifies the plan or the program text after the programmer has completed the modifications. Lastly, at any given time, the programmer is allowed to directly interact with the underlying programming environment.

Despite its limited success in terms of industrial applications, KBEmacs demonstrated a few novel ideas in semi-automated program synthesis:

- a new formalization, plan calculus, which combines existing formalizations,
- a semi-automated system that aims to actively assist the programmer as well as providing freedom to the programmer, and
- a facility for documenting programs, which includes high-level information.

The main shortcomings of KBEmacs and perhaps the programmer's apprentice project are the lack of a strong reasoning system and standard guidelines for division of labor between the programmer and the system. In KBEmacs, in principle, the programmer can altogether bypass the system. However, a set of guidelines may contribute to better division of labor and contribute to stronger reasoning system.

2.1.3 Draco

The Draco project started in the late 1970's with the aim to provide assistance in software construction [32, 33]. Unlike the other approaches to semi-automated program constructions, such as TI and PA, the Draco approach did not intend to advance knowledge representation schemes, planning, language design, parser generation, program transformation, or module interconnection languages. Instead, it focused on combining the successful techniques in each of the mentioned domains into a framework that concentrated on the construction aspects of software.

The designers of Draco intended to drastically reduce the number of development team members needed to produce a large software system. This reduction of number of team members could facilitate a rapid development approach by allowing feedback cycles involving the original specifiers [33]. To achieve this goal, the Draco approach focused on domain specific software systems. Hence, the underlying assumption of this approach is that numerous similar systems would be constructed over time in a given application domain [15].

The Draco's view of software design differs from the traditional view with respect to the notion of domain analysis. In the traditional view there are two phases in software design: the *analysis phase*, where the focus is placed on "what" the system should do, and the *design phase*, where the focus is placed on "how" the system is to perform its function. In Draco's view, there are also two phases. However, these phases focus on a domain of application, as opposed to one single system. The two phases in the Draco approach are [33]:

1. *application domain analysis*, during which the requirements of a collection of similar systems are examined. This phase results in a set of objects and operations encapsulated as a *domain*.

2. *domain analysis*, during which different implementations for the objects and operations of a domain are specified in terms of other known domain to Draco.

As a result, the heart of the Draco approach is a library of problem domains.

In Draco, different application domains are described using a *domain language*, which are defined and implemented in Draco [15]. Once a set of Draco domains has been defined, new systems can be considered in the light of existing system. When a problem is cast as a program of a domain language,

the designer interacts with Draco to refine the problem into executable code. The refinement process is carried out by using a set of transformation rules (in the order of 1000's). These transformation rules, unlike those of TI, are source-to-source rules which only optimize the domain language programs [33]. To limit the number of decisions that the designer has to make, Draco provides a set of domain-independent rules, called *refinement tactics*, for making refinement decisions about the representation and structure of programs [32]. To further reduce the number of possibilities, Draco annotates all program fragments with all possible applicable transformations.

The role of Draco, similar to that of the Programmer's Apprentice [47], is active in the refinement process. The designer never suggests transformations; instead, the designer solicits suggestions from Draco [15]. Draco, in turn, suggests possible transformations based on program annotations and refinement tactics. Once a transformation is applied, it suggests other applicable transformations.

Draco, like other transformational systems, captures the design history by recording the refinement history. The refinement history enables Draco to provide replay mechanisms and also helps in a better understanding of the system [32]. However, this history is very large —Neighbor estimates the history to be ten times the size of the resulting code [33]— and does not capture design rationales. The motivations and rationales are to some degree implicit in the domain where objects and operations are annotated by implementation techniques or applicable transformation rules.

In summary, Draco enables the definition and implementation of languages of various types, provides transformation rules for source to source transformation, and consistency checking mechanisms for transformations. Draco is a semi-automated system that plays an active role in the refinement process, and facilitates the recording of the refinement history [15]. The main obstacle in using the Draco approach is the difficulty of domain analysis, which requires extensive knowledge of a particular application domain.

2.2 Case-Based Systems

Case-based reasoning is a problem solving paradigm, which involves the use of previous experiences (cases) for solving new problems [22]. The motivation behind this paradigm stems from the observation that experts rely more on their previous experiences than a corpus of general knowledge. Furthermore,

as the expert gains more experiences, her problem solving ability will improve.

The main steps of case-based reasoning are:

1. identification of the problem at hand,
2. retrieval of similar cases,
3. adapting the solution in the best matching case(s) to fit the current problem,
4. evaluating the new solution, and
5. storing the new problem and its solution.

An important feature of case-based reasoning is the storage of both positive and failed experiences. Positive experiences can be used to provide problem solving directions, whereas failed experiences can be used to avoid repeating mistakes.

The construction of case-bases is in general simpler than the construction of a traditional knowledge base, which normally consists of general rules. This simplicity is due to the informal records of problem solving experiences that are usually kept by experts. Further, the initial number of cases for an effective case-based system need not be very large so long as they are carefully selected. As case-based systems solve new problems, their case-bases grow and their problem solving abilities, in principle, improve. The main challenges of case-based reasoning, however, are the retrieval of similar cases and their adaptation to fit the new problems. These challenges are not trivial and are among the main focuses of research in case-based reasoning.

Case-based reasoning has a striking resemblance to the compositional software reuse approach as defined by Prieto-Diaz [35]. As a result, in recent years, there has been a growing interest in the research community in applying case-based reasoning to compositional software reuse.

One of the earliest attempts to apply case-based reasoning to software construction was outlined by MacKellar and Maryauski [28]. Their work focused on the construction of a knowledge base for code reuse called *WharfRat*. WharfRat was limited to retrieval of "data types," which are structured definitions of the data and the operations associated with them. These data types are organized in a semantic net with several different types of links: *is a specialization, is an instance of, has a member,* and *has parts* links. The similarity

between data types is also defined by a fuzzy link *is like*. WharfRat is interactive and attempts to retrieve data types by evaluating a similarity score (based on fuzzy logic) to the representation of data types and their links. The goal of this system was to be included in a complete programming-by-similarity system [28]. At the time of this study there were no further information available on the future of WharfRat.

Another early attempt at using case-based reasoning in compositional software reuse focused on program optimization [41]. One of the shortfalls of compositional software reuse is efficiency: composed components sometimes require optimization to meet their performance requirements. The CGS system [41] used case-based reasoning as a learning system, which attempted to facilitate program optimization while preserving correctness of the program. Simply viewed, CGS generates rules for program optimization based on its previous experiences. The process of rule generation is interactive. The system uses its previous experiences to formulate rules that are proposed to the user, who can further qualify them by narrowing the domain of application, or constraining functional and non-functional requirements, etc.

Despite its short history, case-based reasoning has been seriously considered as an enabling technology for software engineering in general and software reuse in particular. More recent attempts in employing case-based reasoning in software construction has focused more on pre-existing components. Furthermore, the degree of automation provided by these systems varies depending on their domain of application. In the following subsection, we look at *Deja Vu* [46], a task oriented program generator, *PROSA* [30], an experimental program synthesis system for educational purposes, *CAESAR* [16, 17] a semi-automatic program synthesis system that uses pre-existing programs from a well-defined application domain to generate new programs, and finally we study the Reuse Assistant [13], which focuses on the retrieval of classes from commercially available class libraries.

2.2.1 Deja Vu

Deja Vu [46] is a case-based reasoning system for constructing software for a task oriented application. The problem domain is the development of plant control software for controlling autonomous vehicles in loading and unloading metal coils in a steel milling process. Deja Vu resembles, in some respect, a very high-level language: it receives a domain specific high-level problem statement (e.g., `Move Two Speed Buggy Forward To Tension-Reel with shop`

`floor layout A`) and produces a high-level solution that can be compiled into an executable program.

In Deja Vu, a case consists of a high-level problem statement, the solution, and a feature set providing a description of the case goals. Cases are stored within a taxonomy that reflects the plant-model planning. These cases can be composite, that is, they can have other cases as part of their solution. This approach allows for greater flexibility during problem solving by providing access to all sub-cases so that the best match for the current problem can be selected.

The novel feature of Deja Vu is its multi-stage problem solving process. Like the expert problem solver, Deja Vu tries to solve problems by successive refinement. This is made possible by composite structure of cases, which can contain other cases at different levels of details. Deja Vu also provides facilities for user interaction during the adaptation process. As mentioned previously, adaptation is the hardest part of reasoning. The interactive adaptation process allows for the human expert to rectify any problems that cannot effectively be dealt with automatically.

2.2.2 PROSA

PROSA is a case-based reasoning system for program synthesis [30]. It was developed to model the learning process in different application domains of programming. Mendiz *et al.* [30] observed that programming is usually taught by examples. Students are typically provided with a set of example programs from which students can abstract program schemas and general programming techniques.

The core component of PROSA is a *Dynamic Knowledge Base* (DKB), which is implemented as a net of frames. The DKB contains two types of information: a basic concept dictionary and a case library. The basic concept dictionary contains the basic concepts in the application domain, which can, at any time, be extended according to the user's needs. Each concept is represented as a set of attribute value pairs. These concepts can be accessed by their names or their attribute lists. The concepts stored in the dictionary are either objects, entities that are manipulated by a program (e.g. strings), or operations, which are actions that manipulate objects (e.g. *length*). The case library in DKB stores cases, which consist of a problem and its solution. The solutions in cases are described at different levels of abstractions to capture the entire refinement process [30]. The final level of a solution can directly

be translated into an imperative programming language. Cases and their subproblems are indexed in DKB by the operations they implement, which allows for reuse of both problem solution and their subproblem solutions in other problems.

When a new problem is presented to PROSA in terms of its attributes, the system searches the DKB for analogous problems by following a top-down strategy. Upon selection of suitable matches, PROSA attempts to interactively and successively adapt the selected solutions to fit the new problem. The main objective of PROSA is educational and suggests the importance of characterizing a problem in terms of domain specific attributes. This suggestion is consistent with the shared knowledge concept introduced in the Programmer's Apprentice project [47].

2.2.3 CAESAR

CAESAR is a case-based reasoning system for constructing programs, in a specific domain, from existing program components [16, 17]. The main goal of CAESAR is to help users to build a first draft of new program from the given specification. To do so, CAESAR contains an adaptation process, in form of Prolog rules, to modify, merge, or extract program fragment from its case base. Furthermore, the cases stored in the case-base contain data tests, which allows CAESAR to evaluate its solution.

The case base in CAESAR consists of programs of the software reuse library, which are composed of the code itself, a repository structure describing the composition of different subparts of the program, and a set of specifications of program subparts and some data test examples. In conjunction with the case base, CAESAR also contains domain knowledge. Domain knowledge is composed of taxonomies representing generalization of concepts and production rules expressing equivalence between specifications.

CAESAR accepts as input a problem specification in the form of a conjunctive of high-level goals. The conjunctive goals are interpreted as a sequence of function calls. CAESAR refines the problem specification using its cases and its domain knowledge. The indexing of cases is accomplished by matching the specifications of program subparts in conjunction with the domain knowledge. The use of domain knowledge relaxes the reliance of the matching process on exact syntactic similarity [36].

The refined specification can be seen as a decomposition of original function calls into a sequence of lower-level function calls. CAESAR tries to find

cases matching these function calls and upon selection of these cases composes them according to its adaptation rules. The next step involves testing of the assembled solution. The results of testing are evaluated and reported to the user for final approval. If approved, the case is added to the case base; otherwise CAESAR repeats its problem solving cycle. It must be noted that CAESAR does not store its failed attempts.

If CAESAR cannot find any cases matching the refined function calls, then it tries to generalize the specification of these function calls. The generalization process is always performed on the data type parts of these specifications and function calls in the user specification always remain unmodified. The generalization results in a different input specification for the selected function. If the generalized specification is accepted by the user, she is required to modify the program code. The new modified specification and program code, after successful testing, is added to the case base.

A novel feature of CAESAR is the seeding of its case base. The initial number of cases in CAESAR does not need to be large. As new problems are solved, the number of cases is increased, which in turn contributes to the problem solving ability of CAESAR: improved efficiency and correctness of solutions. Another novel feature of CAESAR is the use of existing program libraries. Through the use of program understanding techniques, data-flow and control-flow analysis tools, CAESAR can semi-automatically extract properties of programs needed for case representation. The main constraint on program libraries is that they must be small and self-contained encapsulating most of the application domain knowledge. Furthermore, the user must be a domain expert to understand how the library programs must be decomposed in terms of their functional specifications. The domain analysis, though demanding, is a necessity for successful software reuse. CAESAR was tested using a publicly available library of linear algebra routines written in C. The results, though promising, can not be considered as a major success for CAESAR. Linear algebra routines and a few other mathematical domains are well defined and narrow, which are not typical of most application domains.

2.2.4 Reuse Assistant

The Reuse Assistant [13] is less of an automated program synthesis than an intelligent assistant that helps users to locate the best matched component for their purpose. Fernandez-Chazimo *et al.* based their ideas on the observation that in software engineering the required information cannot fully be specified

a priori. Hence, use of traditional information retrieval techniques is insufficient for software reuse; problem setting and problem solving knowledge must be taken into account to help users to articulate their queries. The Reuse Assistant is a tool that allows users to construct their queries incrementally, locate the required components, as well as helping the user in comprehension of the retrieved information.

The Reuse Assistant takes a hybrid approach to storage and retrieval of software components: automatic indexing [42], from traditional information retrieval, to capture syntactic information, and a case-based representation of components to capture the semantic knowledge. Every component in the library is indexed using both approaches. This allows the queries to be expressed either in a restricted natural language based on keyword matching or incrementally by filling a form. The former queries are treated by the information retrieval module, whereas the latter queries are treated by the case base. Furthermore, the results of any queries can be used as an entry point for searching the case base.

The main focus of the Reuse Assistant has been on object oriented languages and commercially available class libraries. Fernandez-Chazimo *et al.* observed that basic libraries cannot be considered at the same level, in terms of reusability, as the user implemented components; these have higher reuse potential [13]. This observation is the justification for hand-coded representation of the user implemented components. First, using statistical methods of automatic indexing a set of keywords are associated with each component. Then, the user can add her own representation as a case. Cases in the Reuse Assistant are implemented as frame like structure with links to other cases. This implementation creates an inheritance hierarchy, where general purpose programming concepts (i.e., methods, classes, and relations) are placed at higher levels and concepts specific to the library and the actual description of the components are placed at lower levels.

The creation of such an inheritance hierarchy requires domain analysis; concepts, objects, operations, and relationships among them must be identified and accurately recorded. As the system is used, new relationships may be added to facilitate future searches. The domain analysis and dynamic update of cases represent the main drawbacks of the Reuse Assistant. The domain analysis is an expensive operation, which requires careful study of the domain. Further, the dynamic update of the cases raises the issues of consistency and the quality of knowledge. Fernandez-Chazimo *et al.* have tested the Reuse

Assistant on a Smalltalk class library and reported a high degree of success. However, the class libraries of commercial languages are generally well designed and conform to the case base structure of the Reuse Assistant. Furthermore, their case study does not answer the problem of consistency and the quality of knowledge. These are open problems that authors reported as being part of their future research [13].

3 A Comparative Study

In this section, we compare the approaches studied in Section 2. This comparison is based on knowledge acquisition, initial cost, the level of support that each method provides, and their overall practicality. Section 3.1, presents a comparison based on knowledge acquisition. We discuss the knowledge representation technique, the type of knowledge used, and how this knowledge is captured. Section 3.2 presents a comparison based on the initial costs involving knowledge acquisition, domain analysis, and the domain of application of each approach. Section 3.3 compares all methods based on their provided support in decision making, evolution process, and learning. Finally, in Section 3.4 we complete our comparative study by taking into account the practical issues. These issues include human-readable documentation, access to the underlying environment, and scalability. Section 3.5 concludes this section by providing an overall ranking for each of the approaches studied.

3.1 Knowledge Acquisition

Knowledge representation is a crucial issue in semi-automatic programming systems. Software engineering is a problem solving activity and as a result knowledge intensive [11]. In order for a semi-automatic programming system to be able to provide assistance in the process of programming, it must reflect the knowledge of the expert. The importance of the knowledge, in turn, raises the issue of the knowledge capture. Table 1 shows a comparison of approaches studied in Section 2 in terms of the type of knowledge that they represent, the type of representation, and how this knowledge is captured.

Rule-based systems rely heavily on reusing production rules with the exception of the Programmer's Apprentice. The Programmer's Apprentice reuses its cliches, in conjunction with rules, to provide higher-granularity reusable artifacts [40]. Case-based systems rely heavily on their cases, which are less

	TI	PADDLE	GLITTER	Programmer's Apprentice	Draco	Deja Vu	PROSA	CAESAR	Reuse Assistant
Knowledge Type	Theoretical	Theoretical	Theoretical/ Domain	Theoretical/ Domain	Theoretical/ Domain	Experience/ Domain	Experience/ Theory	Experience/ Theory	Experience/ Theory
Knowledge Rep.	Production Rules	Production Rules	Production Rules	Cliches/ Rules	Production Rules	Cases	Cases	Cases/ Rules	Cases
Knowledge Acquisition	Formal	Formal	Formal	Formal/ Semi-Formal	Formal	Semi-Formal	Semi-Formal	Semi-Formal	Semi-Formal

Table 1: Comparison based on Knowledge Acquisition

formal than production rules. Cases also provide an advantage that they can offer different levels of abstractions.

The dominant type of knowledge in rule-based systems is *theoretical knowledge*: this is what can be assimilated from a theoretical study of programming [19]. With the exception of the TI and PADDLE systems, other rule-based systems also require some degree of domain knowledge. The case-based systems take a different approach and use mainly experiences or *practical knowledge*: this is what can be obtained from having been active in a practice [19]. Due to the nature of practical knowledge, case-based systems also contain domain knowledge. Experiences are generally domain dependent. CAESAR and PROSA also include some theoretical knowledge, in the form of production rules, to improve the performance of their systems.

The acquisition of knowledge is an important issue in semi-automatic programming systems and directly influences the initial cost of the system. The rule-based systems, due to their reliance on theoretical knowledge, require a formal study of the programming process, perhaps designing a formal language, and formalization of their knowledge in form of production rules. Further, the production rules must be verified for consistency and completeness. This process can be extremely costly, time consuming, and sometimes error-prone, if there is no record of a previous effort that is considered useful.

The case-based systems described use semi-formal approaches to knowledge capture. These systems rely on available records of previously solved problems for their cases. Where such records are available, case-based systems provide a significant advantage in the knowledge acquisition over rule-based systems.

	TI	PADDLE	GLITTER	Programmer's Apprentice	Draco	Deja Vu	PROSA	CAESAR	Reuse Assistant
Initial Cost	High	High	Very High	High	Very High	Medium	Medium	Medium	Medium
Domain of Application	General	General	General/ Specific	General/ Specific	Specific	Narrow/ Specific	Specific	Narrow/ Specific	Specific
Domain Analysis	None	None	Low	Low	High	High	Medium	High	High

Table 2: Comparison based on Initial Cost

3.2 Initial Cost

Another important aspect of semi-automatic programming systems is the initial cost: how much effort must be invested initially with respect to the expected benefits. Most of the automatic systems have some degree of dependence on the domain of application that they are being used for. Table 2 shows a comparison of the approaches studied in Section 2 in terms of their domain of application, the domain analysis efforts, and an initial cost of the system.

Rule-based systems were generally designed with the aim at general programming. They hoped to be applicable to every domain. However, the practical issues with regard to the anticipated support cause some of these systems to include some domain specific information to improve their performance. For example, the Programmer's Apprentice provides a repository of shared knowledge which may contain domain terminologies so that the users do not have to start from first principles [47]. Glitter also provides facilities that require domain knowledge [14]. Hence, in general, rule-based systems have little or no overhead in terms of domain analysis. We must note that the Draco approach, though not domain specific, requires extensive domain analysis [33].

Case-based system, as previously mentioned (see Section 3.1), are in general domain dependent. However, in general, due to the availability of cases and the fact that case-based systems do not require a large number of cases to start, the domain analysis costs are not as high as expected. The Reuse Assistant

	TI	PADDLE	GLITTER	Programmer's Apprentice	Draco	Deja Vu	PROSA	CAESAR	Reuse Assistant
Decision Support	None	Low/ Automatic	Medium/ Semi-Auto.	High/ Semi-Auto.	High/ Semi-Auto.	High/ Semi-Auto.	High/ Semi-Auto.	High/ Semi-Auto.	Low/ Semi-Auto.
Evolution Support	None	Low	Medium	High	High	Very High	High	Very High	None
Learning	Manual	Manual	Manual	Manual	Manual/ Assistance	Automatic	Automaitc	Automatic	Semi-Automatic

Table 3: Comparison Based on User Support

has less overhead in terms of domain analysis since initially it uses automatic indexing for representing its cases. As the system is used, extra efforts must be invested to improve its knowledge-base [13].

3.3 User Support

The goal of semi-automatic programming is to provide support to programmers. As a result, we compare each approach based on the level of support they provide in decision making, evolution, and learning. Table 3 shows a comparison of approaches studied in Section 2 in terms of the support they provide.

Rule-based systems in general do not support learning. They do not record a history of the problems they solve and hence they cannot learn from their experiences. If need be, the programmer must explicitly add new production rules and verify that the newly added rules are consistent with the existing rules. Case-based systems, on the other hand, learn from their experiences: as the system solves more problems, in principle, it becomes more efficient and effective in problem solving.

There are two exceptions to the approaches that we studied. The Draco approach provides assistance in terms of allowing the user to define new domains using the existing domains of Draco [33]. The Reuse Assistant, on the other hand, unlike other case-based systems, does not provide learning. However, it provides users with the ability to reflect what was learned by manually

updating cases [13].

In terms of decision support, we compare each approach based on how much support they provide and how this support is provided. The decision support in the TI system is almost none. The system does not provide any suggestion in terms of what rules can be applied and which ones are more likely a better choice. The PADDLE system provide more support in terms of automatic application of rules. The Glitter system provides more support in terms of allowing the user to interact with the system with regards to the selection of rules and it partially automate the application of these rules. The Programmer's Apprentice and the Draco approach, both provide a higher degree of support in decision making. These systems are active participants and try to make suggestions to the user based on the problem at hand and their knowledge. Furthermore, based on the decisions made, they provide some degree of automation by performing redundant tasks [33, 40].

The degree of decision support in case-based systems is much higher. They semi-automatically refine the input specification, retrieve similar cases, select cases, and perform adaptation. An exception to these systems is the Reuse Assistant, which provides a lower degree of support. It provides only incremental query formation and component location.

Software evolution is increasingly becoming a crucial issue in software engineering: software systems increasingly outlive their life expectancies; their original developers, in most cases, are unavailable, and the knowledge gained during the development process is locked in their minds. The TI system was pioneer in providing support for software evolution. Though it does not provide any support itself, it has influenced other systems in evolution support. The PADDLE system provides a higher degree of support in evolution than TI. However, this support is minimal in comparison to other systems. The PADDLE system records the development process as a sequence of applications of production rules. It does not provide any support in terms of decisions made and the rationales behind them. The Glitter systems goes one step further and attempts to record these lost rationales. However, the decisions stored are not completely machine usable. Further, the above mentioned systems do not provide user readable outputs (see Section 3.4). The Draco approach and the Programmer's Apprentice focus on producing user-readable documentation as well as an internal record of the development history.

Case-based systems generally provide some degree of support in the evolution. Their degree of support provided is dependent on the structure of their

case bases: a case reflects the development history in the form of a hierarchy of refinement steps, where each step can be a case itself. The only exception to this category is the Reuse Assistant, which does not store the history of the development and as a result can not provide much support in the software evolution.

One of the shortcomings of rule-based systems in providing evolution support stems from the lack of information of unsuccessful attempts. To some extent, this is also a reason for the lack of learning ability in such systems. Unsuccessful attempts are crucial in learning: they can be used to reduce the possibility of future failures. With the exception of the Reuse Assistant, all of the other case-based systems presented record both positive and negative experiences, which can be used as a means of support in the evolution process.

3.4 Overall Practicality

Overall, for a semi-automatic programming system to be of practical use, it must provide mechanisms to improve the programming task, from initial specification through evolution. Furthermore, it must not enforce "policies." A system must 1) allow the user to access the underlying environment, 2) allow the user to take full control of the development process, and 3) provide human-oriented documents as well as an internal representation of the development process. Further, a system designed for practical use must also be scalable: prototypical systems are not always an indication of practical success. Table 4 shows a comparison of Section 2 approaches in terms of overall practicality.

The creation of human readable documents is crucial since it allows the human expert to verify or understand what the system has done and what are the reasons behind these decisions. The only system that provides good human readable documents is the Programmer's Apprentice. The Programmer's Apprentice helps in both program commenting and external documentation by using its shared knowledge [40]. Draco, Deja Vu, PROSA, and CAESAR provide low level of support in this area, while other systems do not provide "human readable" documents. TI, PADDLE, and Glitter, produce a specification of the system written in GIST, which has been pretty-printed. However, this document requires a great deal of knowledge about GIST and formal languages.

In terms of scalability, all systems are suspect: to continue to provide their expected level of support, these systems must be able to handle larger bodies

	TI	PADDLE	GLITTER	Programmer's Apprentice	Draco	Deja Vu	PROSA	CAESAR	Reuse Assistant
User Documentation	None	None	None	Good	Low	Low	Low	Low	None
Access to Environment	None	None	None	Good	None	None	None	None	Low
Scalability	None	None	None	None	None	None	None	None	None

Table 4: Comparison Based on Overall Practicality

of knowledge. All reported systems have been successfully tested on a few small- to mid-size test cases. However, there is no evidence as how they would perform in the construction of large-scale systems that may include more than one domain of application.

The final issue that we consider is the access to the underlying environment. In order for a system to be usable it must allow the user to move freely from the system to the underlying environment, for example another support system. The only system that provides such facility is the Programmer's Apprentice, where the programmer has the choice between using the knowledge-based editor or the regular editor [47]. Other systems have all failed to recognize the importance of such support.

3.5 Ranking

In this section, we rank the possible usefulness of each approach based on their domain of application, the promised support, and ease of use as *low, medium,* and *high*. The TI and PADDLE systems are ranked *low*: the specification language GIST is hard to use for average programmer and the support provided by these systems does not justify learning of GIST. The PROSA system was mainly designed for educational purposes and its support is very limited in terms of practical applications. As a result we rank PROSA as *low*. The Glitter system provides better support than the TI and the PADDLE systems. However, it shares a common problem with these systems: the specification

language GIST. We rank the Glitter system as *medium*.

The Draco approach despite its domain analysis is ranked as *high* since the domains defined in the Draco system are potentially reused may times that amortizes the overhead of domain analysis. The ranking for the Programmer's Apprentice is similarly *high*. This is mainly because of the little restrictions and significant level of support provided by the system.

The Deja Vu and CAESAR systems are ranked *high*. Both systems provide a high level of support in their narrow domain of application. The Reuse Assistant is also ranked *high* mainly due to its simple structure. Our justification for this ranking stems from the use of existing commercially available class libraries, use of automatic indexing, and mostly because it is much closer to the status quo than other systems. In fact, the Reuse Assistant can be considered as a small but positive step from the more traditional programming approaches to the semi-automated systems.

4 Future Directions

The study presented in this paper reflects some of the requirements for future efforts in semi-automatic programming systems. First and foremost, we must realize that the human programmer is at the center of the program development process. Hence, the system must provide active support in the decision making process as opposed to completely taking control of the process [47]. Second, the system must provide ease of access to the underlying environment, where the human expert can take advantage of other support systems [15]. Third, a record of the development history, including design decisions, the rationales behind them, and failed attempts, is necessary to providing support for software evolution. The software evolution has become a significant issue in software engineering and as Balzer [5] foresaw in the early 1980's and further realized in the 1990's (for example, see Selfridge *et al.* [43]), software development is indeed an ongoing evolution process. Lastly, the domain analysis is more and more becoming a determining success in software engineering [1]. The semi-automated systems are mostly domain dependent. Hence, such systems must also provide support during the definition and analysis of the application domain.

References

[1] P. D. Bailor. Educating knowledge-based software engineers. In *7th Knowledge-Based Software Engineering Conference*, September 1992.

[2] R. Balzer. A global view of automatic programming. In *3rd International Joint Conference on Artificial Intelligence*, Stanford, California, 1973.

[3] R. Balzer. Final report on GIST. Technical report, Information Science Institute, University of Southern California, 1981.

[4] R. Balzer. A 15 year perspective on automatic programming. *IEEE Transactions on Software Engineering*, SE-11(11), November 1985.

[5] R. Balzer, T. E. Cheatham, and C. Green. Software technology in the 1990's: Using a new paradigm. *IEEE Computer*, November 1983.

[6] R. Balzer, N. Goldman, and D. Wile. On the transformational implementation approach to programming. In *2nd International Conference on Software Engineering*. IEEE, 1976.

[7] F. T. Barker. Chief programmer team management. *IBM System Journal*, 11(1), 1972.

[8] T. J. Biggerstaff and A. J. Perlis eds. *Software Reusability*, volume 1 and 2 of *Frontier Series*. ACM Press, 1989.

[9] A. Borgida and M. Jarke. Knowledge representation and reasoning in software engineering. *IEEE Transactions on Software Engineering*, 18(2), June 1992.

[10] T. Cheatham, G. Holloway, and J. Townley. Program refinement by transformation. In *5th International Conference on Software Engineering*, 1981.

[11] B. Curtis. Cognitive issues in reusing software artifacts. In T. J. Biggerstaff and A. J. Perlis, editors, *Software Reusability*, volume 2 of *Frontier Series*, chapter 20. ACM Press, 1989.

[12] J. Darlington and M. Feather. A transformational approach to modification. Technical Report 80/3, Imperial College, London, 1979.

[13] C. Fernandez-Chazimo, P. A. Gonzalez-Calero, L. Hernandez-Yanez, and A. Urech-Baque. Case-based retrieval of software components. *Expert Systems With Applications*, 9(3), March 1995.

[14] S. F. Fickas. Automating the transformational development of software. *IEEE Transactions on Software Engineering*, SE-11(11), November 1985.

[15] G. Fischer. Cognitive view of reuse and redesign. *IEEE Software*, July 1987.

[16] G. Fouque and Stan Matwin. CAESAR: a system for case based software reuse. In *7th Knowledge-Based Software Engineering Conference*, September 1992.

[17] Glies Fouque and Stan Martin. A case-based approach to software reuse. *Journal of Intelligent Information Systems*, 1993.

[18] P. Freeman. A conceptual analysis of the Draco approach to constructing software systems. *IEEE Transactions on Software Engineering*, SE-13(7), July 1987.

[19] B. Goranzon and I. Josefson. *Knowledge, Skill and Artificial Intelligence.* Springer-Verlag, 1988.

[20] S. Henninger. Developing domain knowledge through the reuse of project experiences. *ACM Software Engineering Notes*, April 1995.

[21] S. Henninger and K. Lappala. Finding the right tool for the job. Technical Report UNL-CSE-94-002, Department of Computer Science and Engineering, University of Nebraska-Lincoln, 1994.

[22] J. Kolondner. *Case-Based Reasoning.* Morgan Kaufmann, 1993.

[23] M. Koubarakis, J. Mylopoulos, M. Stanley, and A. Borgida. Telos: Features and formalization. Technical Report KRR-TR-89-4, University of Toronto, 1989.

[24] C. W. Krueger. Software reuse. *Computing Surveys*, 24(2), June 1992.

[25] B. Lientz and E. Swanson. *Software Maintenance Management.* Addison-Wesley, 1980.

[26] M. R. Lowry. Knowledge-based software engineering. In A. Barr, P. R. Cohen, and E. A. Feigenbaum, editors, *The Handbook of Artificial Intelligence*, volume 4, chapter 20. Addison Wesley, 1989.

[27] M. R. Lowry. Methodologies for knowledge-based software engineering. In J. Komorowski and Z. W. Ras, editors, *Methodologies for Intelligent Systems*, Lecture Notes in Artificial Intelligence, LNCS(698). Springer-Verlag, 1993.

[28] B. K. MacKellar and F. Maryanski. A knowledge base for code reuse by similarity. In *13th Annual International Computer Software and Applications Conference*, September 1989.

[29] M. D. McIlroy. Mass produced software components. In P. Naur and B. Randell, editors, *Software Engineering; Report on a conference by the NATO Science Committee*. NATO Scientific Affairs Division, 1968.

[30] I. Mendiz, C. Fernandez-Chazimo, and A. Fernandez-Valmayor. Program synthesis using case based reasoning. 12 IFIP Congress, Poster Session: Software Development and Maintenance, September 1992.

[31] P. Nauer and B. Randell. Software engineering: Report on a conference by the NATO science committee. NATO Scientific Affairs Division, 1968.

[32] J. M. Neighbors. The Draco approach to constructing software from reusable components. *IEEE Transactions on Software Engineering*, SE-10(5), September 1984.

[33] J. M. Neighbors. DRACO: A method for engineering reusable software systems. In T. J. Biggerstaff and A. J. Perlis, editors, *Software Reusability*, volume 1 of *Frontier Series*, chapter 12. ACM Press, 1989.

[34] H. Partsch and R. Steinbruggen. Program transformation systems. *Computing Surveys*, 15(3), September 1983.

[35] R. Prieto-Diaz. Classification of software modules. *IEEE Software*, 4(1), January 1987.

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

[37] C. Rich. Seven layers of knowledge representation and reasoning in support of software development. *IEEE Transactions on Software Engineering*, 18(2), June 1992.

[38] C. Rich and R. C. Waters. Automatic programming: Myths and prospects. *IEEE Computer*, August 1988.

[39] C. Rich and R. C. Waters. A research overview. *IEEE Computer*, November 1988.

[40] C. Rich and R. C. Waters. Formalizing reusable software components in the programmer's apprentice. In T. J. Biggerstaff and A. J. Perlis, editors, *Software Reusability*, volume 2 of *Frontier Series*, chapter 15. ACM Press, 1989.

[41] S. H. Rubin. Learning in the large: Case-based software systems design. In *1991 IEEE International Conference on Systems, Man, Cybernetics*, October 1991.

[42] G. Salton and M. J. McGill. *Introduction to modern information retrieval.* McGraw-Hill, 1983.

[43] P. G. Selfridge, L. G. Terveen, and M. D. Long. Managing design knowledge to provide assistance to large-scale software development. In *7th Knowledge-Based Software Engineering Conference*, September 1992.

[44] D. R. Smith. KIDS: A semi-automated program development system. *IEEE Transactions on Software Engineering*, 16(9), September 1990.

[45] D. R. Smith. KIDS – a knowledge-based software development system. In M. R. Lowry and R. D. McCartney, editors, *Automating Software Design*, chapter 19. AAAI Press, 1991.

[46] B. Smyth and P. Cunningham. Deja Vu: A hierarchical case-based reasoning system for software design. In *10th European Conference on Artificial Intelligence*, 1992.

[47] R. C. Waters. The programmer's apprentice: A session with KBEmacs. *IEEE Transactions on Software Engineering*, SE-11(11), November 1985.

[48] D. S. Wile. Program developments: Formal explanations of implementations. *Communications of the ACM*, 26(11), November 1983.