

Abstraction of Formal Specifications from Program Code

Betty H.C. Cheng and Gerald C. Gannod

Department of Computer Science
Michigan State University
East Lansing, Michigan 48824

Abstract

The use of formal methods in software development is gaining increasing attention as software becomes more important to the operation of critical systems. Verification of formally specified software may be automated. Unfortunately, most existing software has not been formally specified. This paper describes the development of a tool that abstracts formal specifications from program code, using formal methods and object-oriented techniques.

1 Introduction

As software is used increasingly to control critical systems, correctness becomes paramount [7]. Our overall objective is to design and implement a software development environment consisting of tools that will assist a programmer in the construction of large software products, while supporting the use of formal methods in all phases of software engineering. Towards this end, we have developed the automated program synthesis system, SEED, that constructs imperative programs from formal specifications while verifying their correctness [1, 2]. Our experience with the SEED project has reinforced the theory that manipulation of formal notation is amenable to automation, thus motivating the need for formal representations of software.

Several new languages that support procedural and data abstraction have been developed in the past decade. Parallel and distributed processing have become commonplace. Much existing software was written prior to these developments and, of course, was not formally specified. In order to correctly reimplement such software in new languages and on new architectures, we need to be able to extract their specifications from the existing software. Due to the overwhelming volume of existing code, a manual extraction method would be tedious and prone to errors.

This paper describes the development of the tool AUTOSPEC (**A**utomated **S**pecification) that abstracts formal specifications from program code. The abstraction process can incorporate domain-specific information supplied interactively by the user, as necessary.

The remainder of the paper is organized as follows: The abstraction algorithms and a discussion of the

use of formal methods and object-oriented techniques for the development of AUTOSPEC are given in Section 2. Implementation-specific information is given in Section 3. Related work is described in Section 4. Finally, concluding remarks and future investigations are given in Section 5.

2 Formal Specifications from Code

Predicate logic is the target language for AUTOSPEC because its syntax and semantics are well-defined. In our initial study, we focus on the abstraction of formal specifications from the primary programming structures found in imperative languages, namely, assignments, alternatives, and iteratives. Because we are using an object-oriented approach, the system may be easily tailored to accommodate different source programming languages and target specification languages, given their respective grammars.

2.1 Abstraction and Verification

Currently, the input to AUTOSPEC are programs written in Dijkstra's imperative programming language [3], which contains type declarations in the header, sequential statements, and no references to global variables. The output from AUTOSPEC is the original program code annotated with predicate logic assertions. For every logical grouping of annotated code, the system verifies that the code satisfies the generated specifications using techniques formalized by Dijkstra [3] and Gries [5] and implemented in the SEED system. The remainder of this section discusses the theoretical background of the verification process and the abstraction techniques for each of the primitive programming structures.

2.1.1 Review of Formal Methods

A *precondition* describes the initial state of a program, and a *postcondition* describes the final state. For a given postcondition R and a statement S , the *weakest precondition* $wp(S, R)$ describes the set of states in which the statement, S can begin execution and terminate with R true. Dijkstra [3] defined the semantics for a small programming language in terms of wp . The synthesis rules comprising SEED use the semantics of predicate logic expressions to determine which programming structures should be synthesized in order to satisfy input specifications.

The selection of the programming structures is verified by finding *valid* weakest preconditions with respect to the specifications.

If, after the abstraction process, the *wp* is not a valid logic expression, then AUTOSPEC considers the specification to be erroneous and initiates another abstraction process.

2.1.2 Abstraction and Verification Processes

Algorithms for the abstraction of formal specifications and the corresponding verification processes for the programming structures are described below.

Assignment statements.

Given a statement of the form $\mathbf{x} := \mathbf{e};$ and a precondition U , where U is a logical expression, the following annotated code is constructed

```
{U}                /* precondition */
x := e;
{x = e ∧ U}        /* postcondition */
```

After AUTOSPEC abstracts the postcondition for an assignment statement, SEED is used to find the corresponding *wp*. The *wp* of an assignment statement is expressed as $wp(\mathbf{x} := \mathbf{expr}, R) = R_{\mathbf{expr}}$, which represents the postcondition R with every occurrence of x replaced by the expression \mathbf{expr} . In the above example, the *wp* of the assignment is U , which is satisfied by the original precondition.

Alternative statements.

Given a statement of the form

```
if
    B1 → S1;
    B2 → S2;
    ...
    Bn → Sn;
fi;
```

and a precondition U , where $B_i \rightarrow S_i$ is a guarded command, and statement S_i is executed only if the Boolean expression (guard) B_i is *true*, then a specification is constructed of the form

$$((B_1 \wedge \text{post}(S_1)) \vee \dots \vee (B_n \wedge \text{post}(S_n))) \wedge U,$$

where $\text{post}(S_n)$ is the postcondition of statement S_n . For each guarded command, we abstract one disjunct, where the guard B_i is directly included as part of the disjunct. We apply the abstraction algorithms to statement list S_i in order to obtain the remaining logical expressions for the disjunct, $B_i \wedge \text{post}(S_i)$.

The *wp* of the alternative statement is: at least one guard B_i must be *true* and every guard must logically imply the *wp* of its corresponding statement list S_i with respect to the postcondition R . Symbolically, the *wp* is expressed as

$$(\exists i :: B_i) \wedge (\forall i :: B_i \rightarrow wp(S_i, R)),$$

where ‘ $::$ ’ indicates that the range of the quantified variable i is not used in the current context.

Iterative statements.

Given an iterative statement of the form

```
do
    B1 → S1;
    B2 → S2;
    ...
    Bn → Sn;
od;
```

and a precondition U , we must identify the following items and perform the respective verification tasks:

- *invariant (P)*: an expression describing the conditions prior to entry and upon exit of the iterative structure.
- *guards (B)*: Boolean expressions that restrict the entry into the loop. Execution of each guarded command, $B_i \rightarrow S_i$ terminates with P *true*, so that P is an invariant of the loop.

$$\{P \wedge B_i\} S_i \{P\}, \text{ for } 1 \leq i \leq n$$

When none of the guards is *true* and the invariant is *true*, then the postcondition of the loop should be satisfied ($P \wedge \neg BB \rightarrow R$, where $BB = B_1 \vee \dots \vee B_n$ and R is the postcondition).

- *bound function (t)*: an integer expression representing the bound on the number of iterations. If at least one of the guards is *true* and the invariant is *true*, then the number of iterations is bounded below by t ($P \wedge BB \rightarrow (t > 0)$).
- *statements that make progress towards termination (S)*: these statements must decrease the bound function after each iteration. Each loop iteration is guaranteed to decrease the bound function. Formally, this condition is:

$$\{P \wedge B_i\} t_1 := t; S_i \{t < t_1\}, \text{ for } 1 \leq i \leq n,$$

where $P \wedge B_i$ indicates that the invariant P and guard B_i are *true*, the assignment to t_1 represents the statement making progress towards termination, and S_i represents statements necessary to ensure that the invariant P is still *true* after one iteration.

The following steps are followed in abstracting the specification for an iterative statement:

1. The abstraction algorithm begins with the template for a quantified expression of the form

$$(Q i : \text{range}(i) : \text{expression}(i)),$$

where Q represents one of the symbols \forall, \exists, Σ .

2. The quantified variable(s) are determined by scanning the identifiers occurring in guards B_i .
3. The ranges of the quantified variables are determined by finding statements occurring prior to entry into the loop that assign values to the quantified variables and their occurrences in the guards.

4. For each guarded command, the corresponding statement list includes statements that ensure progress towards termination; the remaining statements ensure that the invariant is maintained upon exit of the loop. The postconditions of these statements constitute $expression(i)$.
5. The bound function becomes the difference between the upper bound of the quantified variable and its value during loop iterations.

Figure 1 gives an example containing all three types of primitive programming structures discussed. The program code finds the index of the maximum element of an array. Because the code contains an iterative structure, AUTOSPEC must determine the quantified variable, its range, and the expression that is being quantified. AUTOSPEC applies the algorithm described previously in order to generate the output in the bottom window of Figure 1. Specifically, AUTOSPEC uses the assignments $i := 1$ and $i := i + 1$ and the guard $i \neq n$ to determine that i is the quantified variable, and its range is $(1 \leq i < n)$. The postcondition of the alternative structure is $(b[i] \leq b[k])$. After simplifying expressions, AUTOSPEC constructs the postcondition

$$(Qi : 1 \leq i < n : (b[i] \leq b[k])) \wedge U,$$

where U is the precondition of the code segment given in the source window of Figure 1.

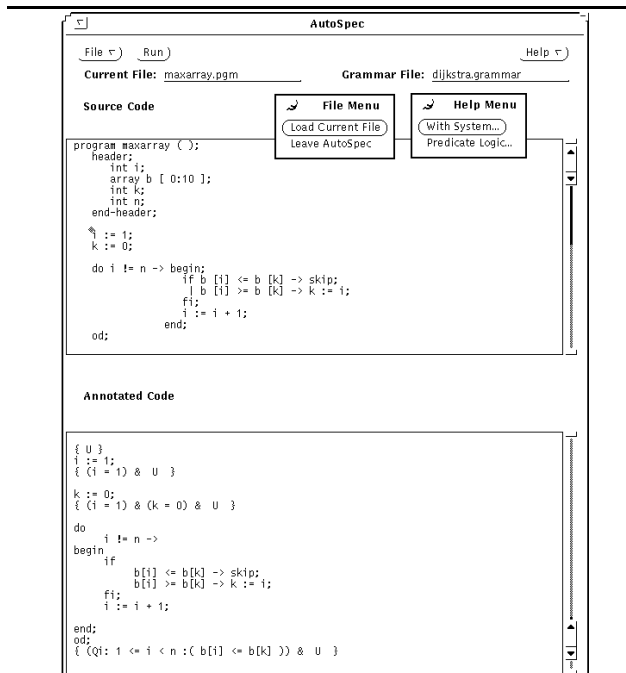


Figure 1: Find the maximum element of an array

2.2 Object-Oriented Development

We developed AUTOSPEC [4] using object-oriented techniques in order to simplify its design, implementation, and maintenance.

We began with an object-oriented analysis involving the identification of a set of classes that exhibit the required behavior of the abstraction system. Input processing tasks include lexical scanning and statement parsing. The abstraction algorithms initially addresses the construction of specification from individually parsed statements and then from logical blocks of program code. For a more detailed description of the classes and their methods, see [4]. Briefly, the main classes are as follows.

typed_element: A *typed_element* is a general base class used for retrieving data from a buffer, where an identification operation is performed on the element contained within the object.

token: *Token* is a subclass of *typed_element* and used to retrieve and identify lexical tokens found in the source code.

statement: A statement is a collection of tokens delimited by semicolons. Allowed types of statements are assignments, alternatives, and iteratives. This class is a subclass of *typed_element* and is used to retrieve and identify statements that are contained in the source code.

specification: *Specification* is a subclass of *typed_element* and is used to retrieve and identify specifications. Its integral purpose is to construct specifications from statements through the use of preconditions and postconditions.

configuration: The configuration class records the current state of facts compiled through the sequential generation of specifications.

queue: A FIFO structure is used between each major class in order to preserve the sequential nature of the source code.

rules: The rules are the guidelines that the system uses to enforce conditions within a language that must be true in order for the program to be syntactically and semantically correct. Subclasses of rules deal with lexical correctness, syntactical correctness, and specification completeness.

3 Implementation

We used formal methods in the design and implementation of AUTOSPEC itself. Due to space constraints, we do not describe the details of the development; see [4] for further information. From a high-level informal description, we specified formally the components of the system using predicate logic [4]. The detailed formal specifications greatly simplified the implementation process, allowing us to focus on issues such as implementation language, target platforms on which to run our system, efficiency, and the user-interface. The remainder of this section describes implementation issues addressed abstraction algorithms and the graphical user interface of AUTOSPEC.

3.1 Implementation Language

We implemented AUTOSPEC with an object-oriented Prolog called LAP¹, which is an integrated

¹LAP is a software product developed by ELSA Software, a company based in Meudon-La-Forêt Cedex France.

system that runs atop Quintus Prolog². LAP offers object-oriented constructs such as classes, methods, inheritance, and polymorphism. A graphical development environment is provided for maintaining hierarchies, instances, and method definitions. Prolog predicates are used to implement all language components.

3.2 Abstraction

AUTOPEC itself was formally specified prior to implementation. The implementation of the abstraction algorithms described in Section 2.1.2 are encapsulated such that each algorithm is a method to a different rule class. Accompanying the classes for the respective statement specification rules are classes that encapsulate the algorithms. We found that there was a significant decrease in time spent on the implementation and maintenance phases of this project as compared to projects of comparable complexity and size that were developed without formal methods and object-oriented design.

3.3 User Interface

Figure 1 contains a screen dump of the graphical user interface for AUTOPEC. Currently, there are two pop-up menus containing Help and File information. The **System** option under the **Help** menu gives descriptions of the input fields, windows, menus, buttons, and actions associated with the mouse. The **Predicate Logic** option provides predicate logic information, (syntax and semantics), definitions of inference rules, definition of weakest preconditions and its role in assuring program correctness. The **File** menu provides access to commands that load files into the **Source Code** window and for exiting from the system. The **Current file** entry is for the name of the source file that is to be processed. The **Grammar File** entry is for the name of the file containing the grammar of the source language; this feature is included in order to allow the system to be tailored for different domains.

When the **Run** button is selected, the source program code is checked for syntactic errors based on the grammar rules supplied earlier; upon a successful syntax check, the annotated program code is displayed in the **Annotated Code** window. Both windows contain scrollbars to allow the user to browse through the input and the output windows.

4 Related Work

[6] Ward et al [9] developed the *Maintainer's Assistant*, which uses a two step process in transforming programs into specifications. First, it restructures code into an equivalent, structured program. Second, the Maintainer, incorporating assistance interactively from the user, constructs a mathematical function that describes the output behavior of the program code. Lano and Breuer [6] have investigated the construction of Z [8] specifications from programs. Their approach involves converting procedural code into a higher level mathematical representation in a sequence of

steps, while using the language of category theory and monads to demonstrate the correctness of the conversion.

5 Concluding Remarks

Availability of appropriate tools contributes much towards the use of methodologies in all sciences today. The same must be assumed in the realm of software engineering and formal methods. The tool described in this paper attempts to bridge the old methods of software development with a more rigorous approach so that the benefits of formal methods can be exploited by existing systems. Considering the ramifications of verifying the correctness of software, the need for such a tool is both important and timely. The design and specification processes provided an opportunity to observe how object-oriented techniques can be integrated with the use of formal methods in the development of software.

Our future investigations will involve the application of AUTOPEC to more examples by expanding the abstraction algorithms. We will also supply grammars of languages such as Pascal and C to construct formal specifications of programs written in the respective languages.

References

- [1] Betty H.C. Cheng. Automated Synthesis of Data Abstractions. In *Proc. of Irvine Software Symposium*, pages 161-176, June 1991.
- [2] Betty H.C. Cheng. Synthesis of Procedural Abstractions from Formal Specifications. In *Proc. of COMPSAC'91*, pages 149-154, September 1991.
- [3] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, New Jersey, 1976.
- [4] Gerald C. Gannod and Betty H.C. Cheng. Formal Specification and Development of AutoSpec. Technical report, Department of Computer Science, Michigan State University, East Lansing, MI 48824, December 1991.
- [5] David Gries. *The Science of Programming*. Springer-Verlag, 1981.
- [6] K. Lano and P.T. Breuer. From Programs to Z Specifications. In J.E. Nicholls, editor, *Z User Workshop*, pages 46-70. Springer-Verlag, Oxford, 1989.
- [7] Mark Moriconi, editor. *International Workshop on Formal Methods in Software Development*, Napa, California, May 1990. ACM SIGSOFT.
- [8] J.M. Spivey. *The Z Notation A Reference Manual*. Prentice Hall International (UK) Ltd., 1989.
- [9] M. Ward, F.W. Calliss, and M. Munro. The maintainer's assistant. In *Proceedings Conference on Software Maintenance*, pages 307-315, Miami, Florida, October 1989. IEEE.

²Quintus Prolog is a product of Quintus Computer Systems, Inc.