

SOFTWARE VERIFICATION RESEARCH CENTRE
DEPARTMENT OF COMPUTER SCIENCE
THE UNIVERSITY OF QUEENSLAND

Queensland 4072
Australia

TECHNICAL REPORT

No. 93-5

A Review of EVES

Jian Chen and Jun Han

May 1993

Phone: +61 7 365 1003

Fax: +61 7 365 1533

A Review of EVES

Jian Chen and Jun Han
Software Verification Research Centre
The University of Queensland, Australia

May 1993

Abstract

We review EVES, a tool for supporting formal mathematical reasoning in general and for the development of formally verified software in particular. The review concentrates on the following three aspects of EVES: its mathematical foundation, its software development method and its automated support. We discuss the strengths and weaknesses of EVES, as well as some issues related to interactive theorem proving and formal method support.

1 Introduction

EVES (Environment for Verifying and Evaluating Software) is a tool for supporting formal mathematical reasoning in general and for the development of formally verified software in particular. It is being developed by ORA Canada. The primary goal of the EVES project is to develop a useful and sound verification system by integrating techniques from automated deduction, mathematics, language design and formal methods.

EVES consists of the following two main components:

- A specification and implementation language called Verdi. The syntax of Verdi is similar to the s-expressions of Lisp. The semantics of Verdi is based on untyped set theory. Verdi is used mathematical modelling in general and for expressing specifications, implementations and proofs of software systems in particular.
- An interactive theorem prover called NEVER, capable of automatically performing large proof steps. Its design was influenced by earlier theorem provers, specifically the Bledsoe-Bruell prover [1], the Stanford Pascal Verifier [15], the Boyer-Moore theorem prover [2] and the Affirm theorem prover [7].

EVES was developed from the same group's earlier experience of prototyping the m-EVES system. An introduction to m-EVES can be found in, for example, [6]. The language and theorem prover of m-EVES are called m-Verdi and m-NEVER respectively. The main differences between EVES and m-EVES are:

- The logical basis of Verdi is a version of untyped first-order logic with a built-in ZFC set theory, while that of m-Verdi is a typed predicate calculus. The main reason for this change is to increase the expressiveness of the language. Typing for executable expressions in Verdi is supported by a mechanism similar to those in programming languages such as PASCAL.
- Recursive procedures are allowed in EVES, but were not supported in m-EVES. This extends significantly the range of problems that EVES can handle.

- NEVER extends the capability of m-NEVER. For example a NEVER user may focus on part of a formula, and has finer control of proof development.
- The EVES documentation and system contain additional applications, for example in hardware, ring theory, abstract data types and critical applications.

The principal designers of Verdi are Mark Saaltink and Dan Craigen. EVES was implemented primarily by Sentot Kromodimoeljo, Bill Pase and Irwin Meisels. The main development was completed in 1990. The EVES project is sponsored by the Canadian Department of National Defence.

Our review is based on the use of the system (version 1.4) and documents [3, 5, 12, 18] provided by the EVES developers, together with some related documents [9, 23]. In this report, we focus on the following issues:

- the mathematical foundation of EVES, including the semantics and underlying logic of Verdi and the proof methods used in NEVER;
- the formal method supported by EVES, including specifications, implementations, proof obligation generation and highly automated interactive deduction;
- system issues, including the language, interpreter, proof commands, library facility and their useability.

The above three aspects are discussed in Section 2 to Section 4, respectively. Each section starts with a brief description which is followed by a discussion of relevant strengths and weaknesses of EVES, as well as our views on some related issues such as interactive theorem proving and formal method support. It should be emphasised that EVES is a system still under refinement and extension. Some of the weaknesses discussed in this report, such as the notation and the user's guide, are among the targets recognised by the EVES developers for further development. It should be noted that a primary goal of the EVES project is to develop a sound and useful system, by integrating state-of-art techniques.

Our review and discussion of EVES is on a rather abstract level. Many details of EVES are not touched. To give the reader a taste of using EVES, in Appendix A we adapt an example from [13], with additional explanation. In Appendix B, we compare EVES with m-EVES based on the discussion in [9].

2 The Mathematical Foundation of EVES

The mathematical foundation of EVES consists of the syntax and semantics of its formal language, the underlying logic and the proof method supported by EVES.

EVES is based on ZFC (Zermelo-Fraenkel axiomatisation with the Axiom of Choice) set theory. The main reasons for choosing set theory, which is based on first-order logic, rather than a type theory, which is based on higher-order logic, are that set theory is well-understood and widely used in computer science, that more is known about automated proofs within a first-order framework than in higher-order systems, and that simple type theory is too restrictive especially for supporting polymorphism.

2.1 The Formal Language of EVES: Verdi

Fundamentally, Verdi is a formal notation based on a version of untyped set theory [3, 18], that can be used to express rigorous mathematical concepts.

Verdi is a first-order language with support for program specification (using pre-conditions and post-conditions) and imperative programming (similar to PASCAL). The syntax of Verdi is similar to that of Lisp s-expressions. The motivation for choosing that syntax is to support an “abstract syntax” for Verdi. This provides the user with possibilities to develop his/her own concrete syntax when using EVES [23]. The version of EVES on which our review is based does not support such concrete syntax extensions. However this has been recognised as one of the targets for further extensions of EVES [4].

Verdi is an untyped language. However typing of expressions which are to be executed is required, through a rather semantic approach. More precisely, support for typing of syntactic forms is provided by using type names, interpreted as sets of values. Verdi has a set of primitive types and type constructors. These are Booleans (Bool), characters (Char), integers (Int), enumerated types, records, arrays and “stub” types. They are used for typing Verdi’s executable functions and procedures.

The EVES development team has demonstrated the expressibility of the untyped set theoretic framework through, for example, the presentation of a combinatory version of Church’s Simple Type Theory [23].

2.2 Verdi Logic

The Verdi logic is a version of the Predicate Calculus with the following features:

1. There are no formulas, only terms. For convenience, terms which play the same role as formulas in conventional logic are often called formulas.
2. Constants are formalised as functions with zero arguments. Every function symbol has a fixed arity.
3. There are definitions, named axioms, executable functions and procedures. The inclusion of procedures requires an extension of conventional logical syntax and semantics: see [18].
4. Many standard notions are used: terms, free and bound variables, substitutions, axioms, theories, proofs and extensions.

There is an EVES initial theory defining all the built-in functions, together with information pertaining to sets (all the ZFC axioms, excluding the schemas), booleans (for example, true, false, not, and, implies, or), integers (for example, the numerals, +, −, div, mod), ASCII characters, elementary functions (for example, <, <=) and arrays.

A Verdi theory consists of a current vocabulary and a set of axioms specifying relationships between the names of the vocabulary. Any theory to be developed in EVES should be a (semantic) conservative extension of the initial theory.

2.3 The Initial Theory of EVES

Although Verdi syntax does not have a reserved identifier notation, the EVES system reserves certain identifiers so that they cannot be redeclared by users. None of the declarations in

the definition of the initial EVES theory have explicit typing information (even for functions that are executable). Such information is known internally to EVES and so is not reproduced by the initial theory. It should be noted that some type checking is part of well-formedness checking in Verdi. Some axioms of the underlying Verdi mathematics [18] cannot be seen from the initial theory since they are represented (and implemented) by other means in EVES.

2.4 Proof Methods

The basic approach of NEVER is to combine automatic proving strategies with interactive user commands. The proof methods used in EVES are based on automatic theorem proving, with support for some human intervention.

A proof in EVES starts from a goal and transforms it into an equivalent form. There can be only one current term as the goal of the proof. If the goal can be transformed into (TRUE) then the proof is successful.

NEVER is neither a fully automatic theorem prover nor a proof checker. It uses many automatic theorem proving techniques to perform large proof steps and at the same time, allows the user to interactively guide the proof. There are four types of automatic proof step which a user can apply during a proof, as follows.

1. *Simplification.* An attempt is made to reduce an expression to one that the system considers simpler. Propositional tautologies are always detected automatically. In addition, simplification reasons automatically about equalities, integers and quantifiers.
2. *Rewriting.* The rewriting process consists of applications of rules. While traversing a term being rewritten, the theorem prover tries to apply rewrite rules that match the subexpression being traversed. A rule may be conditional. The theorem prover must then prove the condition as a subgoal before the rule can be applied.
3. *Reduction.* Reduction is the main strategy used by the automatic theorem prover. Reduction may cause function definitions to be expanded. Non-recursive functions are always expanded (unless disabled). Heuristics developed by Boyer and Moore [2] are used in the case of recursive functions.
4. *Induction.* The theorem prover performs induction using an extension of the technique of Boyer and Moore. Normally, induction schemes are heuristically chosen based on calls to recursive functions within the current formula. It is possible for a user to indicate explicitly on which subexpression to induct. It is also possible to induct on a term (formula) which is not in a formula.

The above four types of proof steps can be seen as combinations of primitive (small) proof steps. NEVER also provides commands to perform a proof manually on a lower level. Each such lower level proof command performs a single primitive proof step, such as the use of an axiom or application of a rule.

2.5 Discussion

The formal language of EVES. Verdi is an untyped first-order language whose semantics is based on ZFC set theory. This gives EVES a simple formal language with sufficient expressiveness, as well as a well-established semantics. There are at least two benefits of this

choice. One benefit is that the soundness of the formal method supported by EVES is relatively easy to guarantee. The other is that the proof system NEVER can be based on many well-established automatic proof techniques.

Formalisations of the language and proof obligations of EVES. As mentioned earlier, developing a sound system was a primary goal of the EVES project. One of the strengths of EVES is that the mathematical foundation underlying its formal language and its notion on proof obligations is well developed [18]. There is also a complementary formalisation of the executable portion of Verdi [19].

Support for typing of syntactic object expressions. In a language for specification and verification of programs, it is important to be able to specify and reason about the type information of program constructs. Although is an untyped language, Verdi supports typing of certain syntactic object expressions by using type predicates in forming domain constraints. This approach overcomes the restrictions inherent in a language with a conventional type syntax.

Heuristic reasoning support and large proof steps. For a formal method tool to support software development, it is important to support suitable and effective techniques for large and complicated reasoning tasks. NEVER combines many heuristic reasoning strategies developed in automatic reasoning. Using heuristic proof support, NEVER is able to carry out large proof steps (such as reduction and induction), as well as primitive proof steps (such as the use of a rule or an axiom). User-defined heuristics are not however supported in EVES. We believe that such support is useful, as it provides the opportunity for the user to build additional capabilities.

Support for interactive theorem proving. For large and complicated reasoning tasks, automatic theorem proving alone is not sufficiently intelligent. Thus interactive support is a necessary feature for practical proof systems. Equipped with many automatic theorem proving techniques, NEVER also has support for the user to guide proofs through a set of commands. However, there is little support in NEVER for composing new proof commands from existing ones.

In the following we discuss, with respect to EVES, two features which appear in some other systems supporting formal methods. However it should be noted that neither feature is intended to be supported in EVES [4].

Support for forming new logics. There are many approaches to the formal development of software. The underlying logics of these approaches vary from first-order to higher-order, and even to non-classical logics. It is therefore an advantage for a formal method tool either to support various logics directly or to support constructions of various logics, from a relatively general basis.

One possible difficulty for supporting more than one logic in EVES would be that the theorem proving component NEVER has many built-in automatic theorem proving techniques, which are specific to the logic currently supported.

Typing support for expressions in a formal language. Typing of syntactic expressions improves the expressiveness of the languages and can improve the efficiency of reasoning. Efficiency of reasoning is typically gained by effectively checking the consistency of typing constraints and hence avoiding redundancy in the reasoning process.

In EVES, typing support for executable expressions is provided for purposes such as determining whether a program construct is defined. However EVES does not support user-defined typing of expressions in general.

As EVES is concerned with software specification and verification, as well as software implementation, it would be useful to extend typing support to non-executable expressions. This would improve the expressiveness of all expressions and hence effectively help all stages of software development, including implementation. This extension would also create possibilities for achieving more efficient reasoning.

3 The Formal Method Supported by EVES

Here we consider EVES support for issues such as program specifications, program implementations, generation and discharge of proof obligations.

Within the method component of EVES, the language Verdi provides ways to program a software system and state its formal specification. The conformance of the software system to its specification is captured by verification obligations. Based on sound theories (conservative extensions of set theory), verification obligations can be discharged via interactive theorem proving using NEVER.

3.1 Specification and Implementation

Verdi is a combined specification and programming language. Its programming mechanisms are executable, and have a style similar to those of imperative programming languages such as Pascal, with some more general programming constructs. The specification mechanisms are not executable, and are partly embedded in the program mechanisms as annotations and partly stated via definitions and properties involving these definitions. The embedded specification features include assertions, preconditions, postconditions, loop invariants and loop termination conditions.

A Verdi program is composed of a sequence of *declarations* which state the specification and programming features required. A declaration may be a type declaration, a function declaration, an axiom declaration or a procedure declaration. The sequence of declarations in a Verdi program should be a conservative extension of the Verdi initial theory.

As explained earlier, there are a number of primitive types and type constructors in Verdi. They are used for typing executable functions and procedures.

A Verdi function is either an untyped function, a ZF function, a recursive function, or a typed function. Typed and untyped function stubs may also be declared. The typed functions and typed function stubs are executable constructs. Verdi functions are defined by expressions rather than by statements so that they have no side effects. They are used for both specification and programming. A typed function or typed function stub has a return type specified and possibly a precondition stated.

An axiom declaration restricts the possible interpretations for the names in the theory vocab-

ulary [3]. It can be seen as part of the system specification. For theorem proving purposes, axioms may be stated in special forms as rewrite rules, forward rules or assumption rules.

A procedure declaration introduces a procedure and is always executable. Procedures may be defined via direct or mutual recursion. A procedure is a program unit similar to a Pascal procedure, but possibly with the following specification annotations: a precondition, a postcondition, and a measure for termination. Procedure stubs may also be declared.

The procedure body is composed of a sequence of statements. The Verdi statements are exit (from a loop), return (from a procedure), abort (the program), note (a form of annotation), assignment, procedure call, block, conditional, case, loop, and two forms of for-loop. The note statement is not executable, but asserts the truth of an expression at its position in the procedure's statement sequence. The loop statement may have an invariant and a termination measure. The for statement may have an invariant. Assertions, invariants and measures are specification features.

3.2 Proof Obligations

As declarations are introduced, the current EVES theory is extended. These declarations result in proof obligations to assure that the conservative extension property holds. The proof obligations are generated automatically by EVES. For recursive function declarations, the resulting proof obligations are to show that some measure, associating natural numbers with recursive calls, decreases with each call.

Proving the verification obligations of procedures and typed functions can be seen as verifying that the system implementation satisfies the system specification. Procedures, typed functions and their verifications use types, axioms, axiomatic rules and non-executable functions; to ensure that the development is logically sound, all these types, axioms, rules and functions have their own verification obligations which must be discharged.

3.3 Structuring Mechanisms

A development in EVES is structured based on Verdi declarations and the use of declarations in other declarations. First, declarations are used to structure specifications and programs. For example, a procedure may be called in another procedure; a function may be used in an expression of another declaration. Second, Verdi declarations also facilitate structuring of proofs in the sense that a declared axiom may be used in the proof of another declaration's verification obligation.

Another structuring mechanism provided by EVES is its support for libraries. A library is a repository of previous developments. Such a development is referred to as a unit, and may be a specification unit, an implementation unit or a saved EVES session (a "frozen" unit). Specification units may be used when carrying out new developments. Procedures and typed functions in implementation units may be used by (i.e., executed from) other developments, if these units' corresponding specification units are used in the latter developments. Therefore, (the specification and implementation units of) EVES libraries provide a mechanism for information hiding, abstraction, modularity and reuse.

3.4 Discussion

Automatic proof obligation generation. One of the key components of EVES is to generate proof obligations based on specifications and implementations of programs, and then prove these obligations. For verifying real-life programs, such a task is often large in size and perhaps tedious. Human direction for discharging proof obligations is needed at critical stages.

EVES proof obligations are automatically generated, once specifications and implementations are provided. This feature of EVES brings advantages, such as increased productivity of software development, as automatic generation is faster than manual work and less error-prone.

Support for theory consistency checking. Checking the consistency of a nontrivial theory for the specification and verification of software is a large and complex problem. Consistency checking is important as an inconsistent theory is worthless. More importantly, when users attempt to form a theory during a formal development of software, they should be helped to find potential problems easily and effectively.

Thus the EVES support for theory consistency checking certainly is a strength. It checks whether a new theory is a conservative extension of the EVES initial theory. The checking is realised by discharging a set of automatically generated proof obligations that ensure the extension conditions.

Support for conventional programming. The EVES approach can be seen as based on conventional (procedural) programming with additional mathematical arguments for program correctness. It represents a “gentle switch” from conventional software development to rigorous software development, which may facilitate industrial acceptance.

Specification and implementation within a unified language. Although specification and implementation are different aspects of software development, reducing the gap between these two components of a formal method support system brings benefits. First, the design of the abstract languages for specification and verification, and the development of their semantics, are simplified. Second, users need less time to learn these languages.

The formal language Verdi is designed for both specification and verification. It achieves this goal by having two types of construct: non-executable functions which provide a functional specification, and executable functions and procedures which provide a software implementation. This approach is particularly effective when combining the implementation of a procedure with the specifications of its pre-conditions and post-conditions.

Post-verification and program refinement. Post-verification and program refinement are two dominant strategies for developing verified software. The former is biased towards verifying the correctness of an existing program while the latter emphasises a stepwise development of a verified software system from its top-level specification, with verification obligations discharged in the development process.

EVES supports program verification by embedding specification annotations in programming constructs. Verification obligations are automatically generated to capture the conformance of the implementation to the specification.

In EVES, one may also carry out software development in a stepwise refinement style. An abstract specification of a software system can be developed using Verdi's non-executable constructs. Then more concrete specifications and implementation may be developed. The conformance between two consecutive levels can then be verified. The library mechanism plays an important role in this development process by providing abstraction and information hiding. However, more documentation of this process and more direct system support for the stepwise development methodology, especially the development path, would be beneficial.

Validation. An important component of a formal method for software development is its support for validation of a software system against its informal requirements. One way to achieve this is to state formal properties (called validation obligations) *expected* of the system and then to prove that the system specification satisfies them. It is certainly possible for EVES users to state validation obligations, and to prove them.

To validate a system specification in EVES, one loads the specification unit from a library and states the expected properties of the system as axioms. Discharging these axioms proves that the system specification satisfies the expected properties. Note that stating the expected properties as axioms within the specification unit might not achieve the desired effect, because one will not be able to tell if a stated property is a further constraint on the names in the theory vocabulary – part of the system specification, or a derivable consequence of the system specification. More explicit documentation of the libraries' role in system validation and the axioms' roles in system specification and system validation would be desirable. A further improvement would be to allow stating and proving, in the system specification, “theorems” which capture derivable properties of the system.

In EVES, the user may also use the `try` command to test (prove) a property expected of the system specification (or the current theory). But the result and proof are not recorded.

Structuring mechanisms. In addition to procedure-based structuring, units in EVES libraries comprise a useful high-level structuring mechanism. Two possible enhancements would be parameterisation of library units, which has already been identified by EVES developers [4], and system-supported inter-library dependencies to allow simultaneous access to units in multiple libraries (see the next section).

4 The EVES System

The EVES system is an environment for specifying, implementing and verifying programs, which supports the formal method reviewed in Section 3 within the logical framework described in Section 2.

In this section, we review various aspects of the design, implementation, use and documentation of the EVES system. We then discuss the strengths and weaknesses of EVES with respect to these system concerns.

4.1 The Language Verdi

In the following list, the expressions in typewriter font are Verdi expressions and the corresponding expressions in italics are in a conventional predicate logic syntax.

$$\begin{array}{ll}
f(x, y) = z & (= (f \ x \ y) \ z) \\
x + 1 < y & (< (+ \ x \ 1) \ y) \\
P(x) \vee Q(y) & (\text{or } (P \ x) (Q \ y)) \\
\forall x \ P(x, y) & (\text{all } (x) (P \ x \ y))
\end{array}$$

A Verdi expression is in one of the following classes: Character Literals, Numerals, Strings, Identifiers, Function Applications, and Quantifications.

Expressions are either executable or non-executable. Executable expressions may be *manifestly* executable, as defined below. An *expression list* is defined as a sequence of expressions.

There are well-formedness rules for Verdi expressions. An expression is well-formed if it satisfies the appropriate syntactic rules and the specific well-formedness rules of the class of expression. A well-formed expression is executable if it satisfies the specific executable well-formedness rules of the class of expression. A well-formed executable expression is manifest if it satisfies the specific manifest well-formedness rules of the class of expression.

Free and bound occurrences of variables are distinguished in Verdi.

For each executable expression, there is a related *legality expression*. This legality expression, when evaluated to the truth value (TRUE), guarantees that the corresponding executable expression can be executed without aborting. The result of executing an executable expression is predictable only if the related legality expression holds. Otherwise the program behaviour is arbitrary.

4.2 Library Support

A library is a repository for objects. The current EVES system supports specification objects, model objects and frozen objects. Specification and model objects provide one form of support for modularisation and abstraction, and frozen objects provide support for saving and restoring the contents of the EVES database. A library object is a representation of the EVES database that was current when the object was created; however, specification and model objects have some restrictions on their creation and contents.

EVES provides library support via commands for: deleting the specified library object, setting the EVES database to specified library objects, loading specified library objects, saving the EVES database to a library, printing the status of a library, and setting the current library.

4.3 The Verdi Interpreter

The interpreter executes procedures and typed functions that have been presented to the EVES system. These procedures and functions may have been entered directly, as top-level declarations, or indirectly, as declarations in library model units whose specifications are loaded into the system.

There are twenty commands that are specific to the Verdi interpreter. These commands can be divided into the following classes based on their functionalities:

- *Set up environments*, including commands for resetting the state of the interpreter, setting interpreter options, and deleting all pending interpretations.
- *Control operations*, including commands for assigning values to specified variables, continuing pending interpretations, declaring and deleting interpreter variables, calling a procedure, and returning from a procedure.

- *Break and trace*, including commands for setting breakpoints in procedures or typed functions, deleting some or all breakpoints, enabling and disabling some or all breakpoints, and enabling and disabling tracing for procedures or typed functions.
- *Display information*, including commands for displaying pending interpretations, names of traced routines, breakpoints, interpreter options, names of cached library model units, the value of an expression, and the values of all interpreter variables.

4.4 Proof Commands

The theorem proving component of EVES is called NEVER. As indicated earlier, the basic approach of NEVER is to combine automatic theorem proving strategies with user interactive commands. There are also commands for displaying information and proof management. The NEVER database contains Verdi declarations, proof obligations and proofs. Partial declarations and proofs are allowed in NEVER.

A proof is performed on a “current formula”, which can be changed by the user. Each proof step aims to transform the current formula into an equivalent formula. The goal of a proof is to transform the current formula to (TRUE). Within a session, the user can suspend the discharge of one proof obligation and move to other proof obligations before continuing this earlier one. A partial proof may also be saved for continuation across sessions.

A set of proof commands is provided in NEVER. Based on their functionalities, we divide them into the following classes:

- *Automatic proofs*. The commands in this class invoke some rather large proof steps based on automatic proof strategies, such as simplification, rewriting, reduction, and induction.
- *Interactive proofs*. Apart from the commands for automatic proof strategies, this class of commands invoke primitive proof steps, under which users have more flexibility in controlling a proof procedure. This includes the use of an axiom or a rule.
- *Formula Transformation*. At some stages, especially before using an automatic proof command, it may be useful to transform a formula to a derived equivalent form. This class of commands is designed for such purposes.
- *Modifying proofs*. NEVER has commands for deleting previous proof steps.
- *Proof management*. This class of commands supports operations such as reading or saving a script file of complete or partial proofs.
- *Environment information*. There are commands which change the current formula, and commands which set states of the NEVER environment.
- *Displaying information*. This class of commands includes commands for displaying help information and commands for displaying information of the NEVER environment, such as proof history, states and proof summary.

4.5 Documentation and Training

The documentation of EVES currently includes *EVES: An Overview* [5], *Reference Manual for the Language Verdi* [3], *A Formal Description of Verdi* [18], *Software Manual for EVES*

Version 1.4 [12], *The EVES Library* [20], *The EVES Library Models* [21], *On-line Help*, and *On-line Examples*. The complete documentation of EVES is still under development [4].

In *EVES: An Overview*, the two components, Verdi and NEVER, are briefly introduced, including their features, structures and theoretical aspects. Several examples are illustrated, though with very limited explanations. The examples are: finding the minimum element of an array, the flow modulator's specification and verification, and the PICO interpreter. An experiment on the possibility of supporting Z specifications in EVES is also discussed.

In *Reference Manual for the Language Verdi*, an informal but complete description of Verdi is given. This description covers the following aspects: the conceptual framework of EVES, the basis of formal syntax of Verdi, components of Verdi (elementary elements, declarations, statements, and libraries), system commands, library commands, interpreter commands, EVES commands. More examples are given in an appendix. There are also appendices for the initial theory of EVES.

A Formal Description of Verdi gives a complete and formal account of Verdi, based on denotational semantics.

In *Software Manual for EVES Version 1.4*, procedures for installing EVES on various machines are briefly described.

In *The EVES Library* and *The EVES Library Models*, a complete description of theories built in the EVES library is given. These theories range from theories for type stub declarations and record type declarations to the theories for ZF function declarations and procedure declarations.

On-line help is provided in EVES. The user can use the EVES command `HELP` to query the functionalities and syntax of many commands, as well as some key concepts of EVES.

There is an example library including specifications, implementations, and proof commands needed. These example files can be read from EVES, by the command `READ`. The EVES will process every line in the file automatically.

A tutorial paper on EVES is being prepared [13].

There is a series of conference papers, for example, [14, 24, 23]. Some other documents on EVES are *Alternative Semantics for Verdi* and *The EVES Library* [19, 20].

ORA Canada also runs several courses on EVES and formal methods. There is a four-day course on EVES and a one-day course on introduction to formal methods [4].

4.6 Extensions to EVES

EVES is being extended in many ways. Two extensions of EVES under development are a rigorous Verdi compiler, and a proof checker [4].

4.7 Discussion

An integrated environment. A formal method tool typically needs to support many stages of the software development process, for example, the specification of theories, the implementation of programs, and the proofs of obligations. Such a system should also be easy to use and should support other system facilities such as libraries, input and output. From the user's viewpoint, an integrated environment including these different system components is desirable.

In EVES, there are many system components, including specification support, implementation support, theorem proving support, library access, and other system support such as on-line help, input and output. These system components are under a unified, easy access user interface. Commands for different tasks can be used without switching from one mode to another.

Development results and processes. EVES supports goal-directed theorem proving based on equivalence transformations. Although most of the proof steps perform transformations, a proof in general has a tree structure because case analysis is allowed. The proof result and the proof script can be recorded, and the proof (script) discharging a verification obligation is associated with the declaration of that obligation. This increases the traceability of developments.

In EVES, the declarations of a development, in which specifications and programs are embedded, are also recorded and can be saved in library objects. As indicated in the discussion of refinement and validation in Section 3.4, however, the EVES method does not adequately address the entire software development process, and hence there is not sufficient corresponding tool support.

Documentation of EVES. The documentation of a formal method tool should include at least a reference manual, user manual, and a formal description of the system. A user's manual is necessary and important for ordinary users of the system. A reference manual provides a complete and, desirably, plain description of the system. A formal description provides a formal justification in terms of the underlying theories.

EVES already has a reference manual, a formal description of Verdi and an alternative description of the executable portion of Verdi. A user's manual is planned [4].

On-line help. On-line help is particularly useful for a large and complex system. For systems like EVES, there are many notions and various types of commands. Although the current implementation of on-line help in EVES is still limited and needs improvement, the user can use the facility as a quick reference during an EVES session.

Traceable proof process and replay. When a problem is large or complex, the user often needs to attempt many possible methods before achieving the desired goals. Thus facilities to support these tasks have a significant influence on a user's productivity.

EVES allows the user to delete certain previously issued commands and set the (system) environment back to assigned points. EVES also provides a few simple but useful commands for saving all displayed information during a session.

EVES has commands for reading in a sequence of specification statements and commands, and for executing each of them one by one. This provides a simple facility for using a set of commands or specifications more effectively. For example, a user can save a part of a session and re-run it, perhaps with some modifications when a similar situation occurs.

Automation vs. user intervention. The basic philosophy of theorem proving in NEVER is to combine automatic strategies with user intervention (by commands). The user may use command modifiers with some of these capabilities to gain finer-grained control.

Another way to achieve fine-grained user control in NEVER is via commands which use a specific axiom, expand a specific function definition, or apply a specific rewrite rule. There are also commands which can be used to manipulate the current formula with respect to quantifiers, equality and arrangement.

In principle, an interactive system should achieve a balance between automation and user intervention both operationally and structurally. No matter how large a proof step is, NEVER shows only the final result. In some cases, the formula is transformed into a form with little resemblance. Before deciding the next step, the user has to re-understand the formula. This could take some time for a large formula. It would be helpful to record the key transformation steps which are carried out automatically by the system, so that the user may review them if so desired.

There are also cases in EVES where the user might not have as much control as desired over deduction steps. For example, there is a case shown on page 83 of the EVES reference manual [3] where EVES heuristics do not allow effective use of axioms as the user wishes.

Structuring proofs. To structure a large proof using additional lemmas and theorems in EVES, the user has to modify the Verdi program, rather than directly introducing and proving the desired lemmas and theorems away from the program specification and implementation. Experience with other theorem proving systems has shown that allowing proof, recording and reference of user-stated theorems can help structure large proofs, without changing program specifications or implementations.

Structuring system support. In EVES, a session is presumably for one development at a time. Such a development can be saved as a library object. Although one development can be used by another development by loading library objects, the libraries are managed separately from the EVES database and only one library is accessible at a time.

Allowing simultaneous access to objects in multiple libraries and providing direct support for inter-library dependencies would be two natural extensions to the library organisation mechanism in EVES. Based on the idea that a library characterises an application domain, inheritance and morphism relations between libraries could be introduced to capture inclusion and similarity relations between application domains (although depending on the degree of support desired, this may be not easy). This would facilitate consistency maintenance, and achieve greater and more flexible reuse. To a certain extent, this idea has been experimented with by some other interactive theorem proving systems, including HOL [8] and Mural [11], where structuring of theories extends the capabilities of EVES libraries. Experience has shown that these features are useful in structuring system support and in achieving flexible reuse while maintaining logical consistency.

Some useability issues of the EVES system.

- *Error tolerance.* EVES is intended to be interactive. However, the system does only limited checking on command syntax. Some Lisp syntax errors and file path mismatches cause EVES to enter to the Lisp Debugging Environment. An ordinary EVES user must become familiar with that environment, in order to cope with normal EVES use. This is regrettable.
- *Robustness.* The system's behaviour appears to be history-dependent. For example, if no Lisp Debugging Trap has been executed during an EVES session, the command (QUIT)

will successfully terminate the EVES system. However we noticed that once a Lisp Debugging Trap has been executed, the command (QUIT) no longer works satisfactorily. The only way we have figured out to terminate the system is by the Lisp system command (LISP:BYE).

- *On-line help.* EVES provides on-line help. However the provided information is sometimes not sufficient. The current help information tends to be focused on declarations, proof commands and related issues, and is less informative on system manipulation matters. For example, when we could not terminate EVES by using (QUIT), the answer to our request for on-line help was: `The topic QUIT has no help.`
- *Documentation.* The current EVES documentation is not easy to use. For example, the Reference Manual is not quite complete; we could not find the commands (QUIT) and (READ string). (They may be commands of the underlying Lisp system; but as necessary parts of using EVES they should be documented in its Reference Manual.) Without such information, a new user feels lost in using the EVES system. We also feel that a User's Manual is indeed necessary, with sufficient examples and explanations of the usage of all commands and constructs (whether or not they are inherited from Lisp).
- *Proof display.* When a proof step occupies more than one screen or the proof process is directed by a ".ver" file, it is difficult for users to follow the display.
- *Notations.* Users from a non-Lisp background may feel not easy in reading EVES expressions, which is similar to Lisp's s-expressions. The EVES group is working on support for user-defined syntax [4].

5 Conclusions

In this report, we have reviewed the EVES formal method tool from a user's viewpoint. The mathematical foundation of EVES, the basic features of the EVES method and the computer-aided support for the method have been examined. Their strengths and weaknesses have been discussed.

In summary, EVES is a usable and powerful system for developing verified software, based on the traditional verification condition generator approach. In particular, its automatic theorem proving capabilities are impressive. Some of the possible areas for improvement include: more direct support for top-down refinement of specifications to code, more structuring mechanisms for system development, the addition of a user manual, and more comprehensive search commands. As noted throughout this document, some of these areas are already being addressed.

6 Acknowledgements

Thanks to Dr. Brian Billard, of the Information Technology Division of the Australian Defence Science and Technology Organisation, for arranging access to the EVES system. Thanks to ORA Canada for their support in carrying out this review. Thanks to Dan Craigen, Mark Saaltink, Sentot Kromodimoeljo and Bill Pase from ORA Canada, and Tony Cant and Maris Ozols from the Information Technology Division of the Australian Defence Science and Technology Organisation, for their comments on earlier versions of this report.

We thank John Staples and Jim Welsh for their advice, support and encouragement. John Staples read several earlier versions of this report and made many useful comments, in particular on the presentation of this report. Comments from our discussions with Peter Kearney, Ray Nickson, Owen Traynor and Mark Utting of SVRC are also helpful to the preparation of the final version.

References

- [1] Bledsoe, J.A. and Bruell, P., “A Man-Machine Theorem Proving System”, in *Proceedings of the Third IJCAI*, Stanford University, 1973, also in *Artificial Intelligence* 5(1):51-72, 1974.
- [2] Boyer, R. and Moore, J.S., “*A Computational Logic Handbook*”, Volume 23 of *Perspectives in Computing*, Academic Press – Harcourt Brace Jovanovich, Boston, United States, 1988.
- [3] Craigen, D., “*Reference Manual for the Language Verdi*”, EVES Project TR-91-5429-09a, 156 pages, Odyssey Research Associates, 265 Carling Avenue, Suite 506, Ottawa, Ontario K1S 2E1, Canada, February 1990 (latest revision date: September 1991).
- [4] Craigen, D., et al, personal communications, 1993.
- [5] Craigen, D., Kromodimoeljo, S., Meisels, I., Pase, B. and Saaltink, M., “EVES: An Overview”, in *Proceedings of VDM '91*, Noordwijkerhout, The Netherlands, October 1991.
- [6] Craigen, D., Saaltink, M., Pase, B., Kromodimoeljo, S., Meisels, I. and Nielson, A., “m-EVES: A Tool for Verifying Software”, in *Proceedings of the 10th International Conference on Software Engineering*, Singapore, April 1987.
- [7] Gerhart, S.L., Musser, D.R., et al., “An Overview of AFFIRM: A specification and verification system”, in *Information Processing 80, Proceedings of IFIP Congress 80*, Lavington, S.H. (ed.), volume 8 of *IFIP Congress Series*, pages 343-347, North-Holland Publishing Company, October 1980.
- [8] Gordon, M.J.C., “HOL: A Proof Generating System for Higher-Order Logic”, in *VLSI Specification, Verification and Synthesis*, Birtwistle, G. and Subrahmanyam, P.A. (eds.), pages 73-128, Kluwer Academic Publishers, Boston, Massachusetts, 1988.
- [9] Grundy, J., “Report on m-EVES”, Research Report ERL-0545-RR, 43 pages, Electronics Research Laboratory, Information Technology Division, DSTO Australia, March 1991.
- [10] Grundy, J., “Window Inference in the HOL System”, in *Proceedings of the International Tutorial and Workshop on the HOL Theorem Proving System and Its Applications*, Windley, P.J., Archer, M., Levitt, K.N. and Joyce, J.J. (eds.), University of California at Davis, August 1991, ACM/IEEE, IEEE Computer Society Press, 1992.
- [11] Jones, C.B., Jones, K.D., Lindsay, P.A. and Moore, R., “*mural: A Formal Development Support System*”, Springer-Verlag, London, 1991.
- [12] Kromodimoeljo, S., Meisels, I. and Pase, B., “*Software Manual for Eves Version 1.4*”, 6 pages, ORA Canada, Ottawa, Ontario K1S 2E1, Canada, March 1992.

- [13] Kromodimoeljo, S., Pase, B., Saaltink, M., Craigen, D. and Meisels, I., “A Tutorial on EVES”, Draft, 45 pages, Odyssey Research Associates, 265 Carling Avenue, Suite 506, Ottawa, Ontario K1S 2E1, Canada, November 1992.
- [14] Kromodimoeljo, S., Pase, B., Saaltink, M., Craigen, D. and Meisels, I., “The EVES System”, in *McMaster International Lecture Series on Functional Programming, Concurrency, Simulation and Automated Reasoning*, Peter Lauer (ed.), Workshops in Computing, Springer-Verlag, to be published in 1993.
- [15] Luckham, D.C., et al., “Stanford Pascal Verifier User Manual”, Report STAN-CS-79-731, Computer Science Department, Stanford University, March 1979.
- [16] Parnas, D. and Bartussek, W., “Using Traces to Write Abstract Specifications for Software Modules,” in *Information Systems Methodology, Proceedings of ICS*, Lecture Notes in Computer Science 65, Springer-Verlag, 1978.
- [17] Parnas, D., Smith, D. and Pearce, T., “Making Formal Software Documentation More Practical: A Progress Report,” Technical Report 88-236, Department of Computing and Information Science, Queen’s University at Kingston, November, 1988.
- [18] Saaltink, M., “A Formal Description of Verdi”, EVES Project TR-90-5429-10a, 78 pages, Odyssey Research Associates, 265 Carling Avenue, Suite 506, Ottawa, Ontario K1S 2E1, Canada, October 1989 (latest revision date: November 1990).
- [19] Saaltink, M., “Alternative Semantics for Verdi”, ORA Canada Technical Report TR-90-5446-02, November 1990.
- [20] Saaltink, M., “The EVES Library”, EVES Project TR-91-5449-03, Odyssey Research Associates, Ottawa, Canada, 1991.
- [21] Saaltink, M., “The EVES Library Models”, EVES Project TR-91-5449-04, Odyssey Research Associates, Ottawa, Canada, 1991.
- [22] Saaltink, M., “Z and EVES: A Summary”, in *Proceedings of the 6th Annual Z User Meeting*, York, England, December 1991.
- [23] Saaltink, M. and Craigen, D., “Simple Type Theory in EVES”, in *Proceedings of IV Higher-Order Workshop*, Banff, Alberta, Canada, 10-14 September, 1990, pages 218-244.
- [24] Saaltink, M., Kromodimoeljo, S., Pase, B., Craigen, D. and Meisels, I., “An EVES Data Abstraction Example”, in *Proceedings of FME’93 (Formal Methods Europe)*, Odense, Denmark, April 1993.

- The function `left` returns the sequence of elements to the left of the pointer.
- The function `right` returns the sequence of elements to the right of the pointer.
- The function `count` returns the number of elements in `table/list`.
- The function `ok` returns a boolean value. It constrains the types and the relationship between the two pointers, and in fact is an invariant function of `table/list`.

A set of executable functions is also declared. Some of them are also used in specifications of pre-conditions and post-conditions of other executable functions and procedures. A brief description of these functions is as follows.

- The function `out` returns a boolean value indicating whether there exists a current element, that is, whether the pointer to the current element is after the start of `table/list`.
- The function `current` returns the current element of `table/list`. It should be called only when a current element exists.
- The function `exleft` returns a boolean value indicating whether there exists an element to the left of the current element.
- The function `exright` returns a boolean value indicating whether there exists an element to the right of the current element.

A set of procedures are defined for operations of `table/list`. A brief description of these procedures is as follows.

- The procedure `init` initialises `table/list`.
- The procedure `delete` removes the current element from `table/list`.
- The procedure `insert` adds an element to the right side of the current element, or at the beginning of `table/list` if there is no current element. This new element becomes the current element of `table/list`.
- The procedure `alter` changes the value of the current element of `table/list`. It should be called only when there is a current element in `table/list`.
- The procedure `goright` moves the pointer of the current element to the element on the right side of the current element. It should be called only when there is at least one element to the right of the current element.
- The procedure `goleft` moves the pointer of the current element to the element on the left side of the current element. It should be called only when there is at least one element to the left of the current element.

A.2 The EVES Process for Table/List

We describe the EVES process of specifying and verifying the implementation of `table/list`. Our description is divided into the following four stages: for initialisations, for non-executable functions, for executable functions and for executable procedures.

In the following, lines in typewriter font starting with `>`, and following lines with at least one space at the beginning, are typed by the user. Lines without `>` or spaces at the beginning are output from EVES, except that lines of the form

```
[...]
```

indicate output from EVES is omitted.

A.2.1 Initialisation

To load the theory of sequences, we first set, by the library command `SET-LIBRARY`, the current theory unit to the theory of sequences. On our system, the full file name for the unit is `/u1/solution/ivg/eves/library/seq.spec`. We can then use the library command `LOAD` to load the theory of sequences from the EVES standard library.

```
>(SET-LIBRARY "/u1/solution/ivg/eves/library/seq.spec")
>(LOAD seq)
```

Next we declare a stub for the element type of `table/list` and define the value of the maximum capacity.

```
>(type-stub entry)
>(typed-function capacity () (int) () 50)
```

In response to the declaration of `capacity`, EVES first returns a line containing `CAPACITY`, and then outputs the proof obligation associated with the declaration. (Note that it is allowed in EVES that a user moves to other declarations or proofs without proving this obligation, and may return to this obligation later. The status of the proof obligation remains unproved. Here is the actual output from EVES.

```
CAPACITY
```

```
Beginning proof of CAPACITY ...
(= (TYPE-OF 50) (INT))
```

In this case, one proof command `SIMPLIFY` proves that the obligation is equivalent to `TRUE`, completing the proof.

```
>(SIMPLIFY)
```

```
Which simplifies
forward chaining using ELEM-TYPE-P.TYPE-P
with the assumptions ELEM-TYPE-P.CHAR, ELEM-TYPE-P.INT, TYPE-P.BOOL to ...
(TRUE)
```

This example illustrates that a brief description of the proof process is provided by EVES.

Next the axiom `maxint-exceeds-capacity` is stated. It is used in reasoning about `capacity`. The simple property `in-type` is then used in later stages.

```
>(grule maxint-exceeds-capacity () (>= (int.last) (capacity)))
MAXINT-EXCEEDS-CAPACITY
```

Beginning proof of MAXINT-EXCEEDS-CAPACITY ...

```
(>= (INT.LAST) (CAPACITY))
>(rule in-type
  (x t)
  (implies (type-p T)
    (= (in x t) (= (type-of x) t))))
IN-TYPE
```

Beginning proof of IN-TYPE ...

```
(IMPLIES (TYPE-P T)
  (= (IN X T) (= (TYPE-OF X) T)))
>(USE TYPE-OF.DEFINITION)
```

Assuming TYPE-OF.DEFINITION generates ...

```
(IMPLIES (AND (ALL (X$0 Y)
  (IMPLIES (TYPE-P Y)
    (= (IN X$0 Y) (= (TYPE-OF X$0) Y))))
  (TYPE-P T))
  (= (IN X T) (= (TYPE-OF X) T)))
>(SIMPLIFY)
```

Which simplifies

```
forward chaining using ELEM-TYPE-P.TYPE-P
with the assumptions ELEM-TYPE-P.CHAR, ELEM-TYPE-P.INT, TYPE-P.BOOL
with the instantiations (= Y T)
(= X$0 X) to ...
(TRUE)
```

The type table/list is defined by the following type declaration.

```
>(type table/list
  (record ((store) (array 1 (capacity) (entry))) ((lp rp) (int))))
```

TABLE/LIST

A.2.2 Non-Executable Functions

As mentioned earlier, the function `left` returns the sequence of elements to the left of the pointer. This function calls an auxiliary function `left-aux`, which recursively constructs the sequence representing the elements to the left of the pointer. Because `left-aux` is recursive, a measure is provided. Using the proof command `reduce`, the function `left-aux` is proved to be well-founded. The axiom `seqp-left`, proved by induction, shows that the function `left-aux` indeed returns a sequence. The function `left` can then be defined in terms of `left-aux`.

```
>(function left-aux
  (a i)
```

```

      ((measure i))
      (if (and (in i (int))
              (>= i 1))
          (seq!tack (aref a i) (left-aux a (- i 1)))
          (seq!empty)))

```

LEFT-AUX

Beginning proof of LEFT-AUX ...

```

(IF (AND (IN I (INT))
        (>= I 1))
    (M< (- I 1) I)
    (TRUE))

```

>(REDUCE)

Which simplifies

with invocation of M<

when rewriting with IN-TYPE

forward chaining using >=.SAME.TYPE, ELEM-TYPE-P.TYPE-P

with the assumptions SUCC.INT, SUCC.TYPE-OF, -.TYPE-OF, ELEM-TYPE-P.CHAR,
ELEM-TYPE-P.INT, TYPE-P.BOOL to ...

```

(TRUE)

```

```

>(grule seqp-left (a i) (seq!seqp (left-aux a i)))

```

SEQP-LEFT

Beginning proof of SEQP-LEFT ...

```

(SEQ!SEQP (LEFT-AUX A I))

```

>(PROVE-BY-INDUCTION)

Inducting using the following scheme ...

```

(AND (IMPLIES (AND (IN I (INT))
                  (>= I 1)
                  (*P* A (- I 1)))
        (*P* A I))
     (IMPLIES (NOT (AND (IN I (INT))
                       (>= I 1)))
              (*P* A I)))

```

produces ...

```

(AND (IMPLIES (AND (IN I (INT))
                  (>= I 1)
                  (SEQ!SEQP (LEFT-AUX A (- I 1))))
        (SEQ!SEQP (LEFT-AUX A I)))
     (IMPLIES (NOT (AND (IN I (INT))
                       (>= I 1)))
              (SEQ!SEQP (LEFT-AUX A I))))

```

Which simplifies

```

with invocation of LEFT-AUX
when rewriting with IN-TYPE
forward chaining using SEQ!SEQP-EMPTY, SEQ!TACK-HEAD-TAIL, >=.SAME.TYPE,
ELEM-TYPE-P.TYPE-P
with the assumptions SEQ!TACK-NOT-EMPTY, SEQ!SEQP-TACK, SEQ!HEAD-TACK,
SEQ!TAIL-TACK-SEQ, AREF.TYPE-OF, -.TYPE-OF, SEQ!EMPTY-EMPTY,
SEQ!SEQP-EMPTY,
ELEM-TYPE-P.CHAR, ELEM-TYPE-P.INT, TYPE-P.BOOL to ...
(TRUE)

```

```

>(function left (t) () (left-aux (store t) (lp t)))
LEFT

```

Similarly, functions `right-aux` and `right` are defined, and the axiom `seqp-right` is proved, as follows. For brevity, we omit EVES output about the proof process.

```

>(function right-aux
  (a i)
  ((measure (+ 1 (- (capacity) i))))
  (if (and (in i (int))
           (<= i (capacity)))
      (seq!tack (aref a i) (right-aux a (+ i 1)))
      (seq!empty)))
[...]
>(REDUCE)
[...]
>(grule seqp-right (a i) (seq!seqp (right-aux a i)))
[...]
>(PROVE-BY-INDUCTION)
[...]
>(function right (t) () (right-aux (store t) (rp t)))

```

The rewrite rules `left-aset` and `right-aset` are used for simplification of terms involving `left-aux` and `right-aux` respectively. They show that assignments outside the section of the array do not affect the value. In each of the two cases the proof is by induction.

```

>(rule left-aset
  (a i j v)
  (implies (and (= (type-of i) (int))
                (= (type-of j) (int))
                (= (type-of a) (array 1 (capacity) (entry)))
                (< i j)
                (<= 1 j)
                (<= j (capacity))
                (= (type-of v) (entry)))
           (= (left-aux (aset a j v) i) (left-aux a i))))
[...]
>(INDUCT)

```

Inducting using the following scheme ...

```
(AND (IMPLIES (AND (IN I (INT))
                  (>= I 1)
                  (*P* A (- I 1) J V))
      (*P* A I J V))
 (IMPLIES (NOT (AND (IN I (INT))
                   (>= I 1)))
          (*P* A I J V)))
```

produces ...

```
(AND (IMPLIES (AND (IN I (INT))
                  (>= I 1)
                  (IMPLIES (AND (= (TYPE-OF (- I 1)) (INT))
                              (= (TYPE-OF J) (INT))
                              (= (TYPE-OF A) (ARRAY 1 (CAPACITY)
                                                    (< (- I 1) J)
                                                    (<= 1 J)
                                                    (<= J (CAPACITY))
                                                    (= (TYPE-OF V) (ENTRY))))
                              (= (LEFT-AUX (ASET A J V) (- I 1))
                                (LEFT-AUX A (- I 1))))))
          (IMPLIES (AND (= (TYPE-OF I) (INT))
                      (= (TYPE-OF J) (INT))
                      (= (TYPE-OF A) (ARRAY 1 (CAPACITY) (ENTRY)))
                      (< I J)
                      (<= 1 J)
                      (<= J (CAPACITY))
                      (= (TYPE-OF V) (ENTRY)))
                  (= (LEFT-AUX (ASET A J V) I) (LEFT-AUX A I))))
 (IMPLIES (NOT (AND (IN I (INT))
                   (>= I 1)))
          (IMPLIES (AND (= (TYPE-OF I) (INT))
                      (= (TYPE-OF J) (INT))
                      (= (TYPE-OF A) (ARRAY 1 (CAPACITY) (ENTRY)))
                      (< I J)
                      (<= 1 J)
                      (<= J (CAPACITY))
                      (= (TYPE-OF V) (ENTRY)))
                  (= (LEFT-AUX (ASET A J V) I) (LEFT-AUX A I))))))
>(REDUCE)
```

Which simplifies

with invocation of LEFT-AUX

when rewriting with AREF.ASET, RANGE.DEFINITION, SUCC.INT.RULE, IN-TYPE forward chaining using SEQ!TACK-HEAD-TAIL, >=.SAME.TYPE, ELEM-TYPE-P.TYPE-P with the assumptions SEQ!TACK-NOT-EMPTY, SEQ!SEQP-TACK, SEQ!HEAD-TACK, SEQ!TAIL-TACK-SEQ, AREF.TYPE-OF, -.TYPE-OF, SEQ!EMPTYP-EMPTY, SEQ!SEQP-EMPTY,

```

SEQP-LEFT, ASET.TYPE-OF, +.TYPE-OF, SUCC.INT, SUCC.TYPE-OF, ARRAY.LOB,
ARRAY.HIB, ARRAY.COMPONENT-TYPE, TYPE-P.ARRAY, ENTRY.TYPE-P,
ELEM-TYPE-P.CHAR,
ELEM-TYPE-P.INT, TYPE-P.BOOL to ...
(TRUE)

```

```

>(rule right-aset
  (a i j v)
  (implies (and (= (type-of i) (int))
                (= (type-of j) (int))
                (= (type-of a) (array 1 (capacity) (entry)))
                (> i j)
                (<= 1 j)
                (<= j (capacity))
                (= (type-of v) (entry))))
    (= (right-aux (aset a j v) i) (right-aux a i))))
[... ]
>(INDUCT)
[... ]
>(REDUCE)
[... ]

```

The next two rewrite rules give the length of the sequence returned by the functions `left-aux` and `right-aux` respectively. Both proofs are based on induction. The proof for the second rule `length-right` is however more complicated and requires several commands.

```

>(rule length-left
  (a i)
  (= (seq!length (left-aux a i))
     (if (and (= (type-of i) (int))
              (<= 1 i))
         i
         0)))
LENGTH-LEFT

```

Beginning proof of LENGTH-LEFT ...

```

(= (SEQ!LENGTH (LEFT-AUX A I))
   (IF (AND (= (TYPE-OF I) (INT))
            (<= 1 I))
       I
       0))

```

```

>(PROVE-BY-INDUCTION)
[... ]
>(rule length-right
  (a i)
  (= (seq!length (right-aux a i))
     (if (and (= (type-of i) (int))
              (<= i (capacity)))
         i
         0)))

```

```

      (+ 1 (- (capacity) i))
      0)))
[...]
```

>(INDUCT)

```

[...]
```

>(REDUCE)

```

[...]
```

>(INVOKE RIGHT-AUX)

```

[...]
```

>(REDUCE)

```

[...]
```

The function `count` indicates the number of elements in `table/list`. This is needed in the axiom `ok-guarantees`, which will be introduced immediately after the function `ok`.

```

>(function count (t) () (+ (seq!length (left t)) (seq!length (right t))))
COUNT
```

The function `ok` is an invariant function of `table/list`. It constrains the types and the relationship between the two pointers.

```

>(function ok
      (t)
      ()
      (and (= (type-of t) (table/list))
            (<= 0 (lp t))
            (< (lp t) (rp t))
            (<= (rp t) (+ 1 (capacity)))))
OK
```

The invariant implies a number of useful properties, namely that the functions `left` and `right` return sequences, and that `table/list` does not exceed its capacity. The proof is by using a single command `REDUCE`.

```

>(frule ok-guarantees
      (t)
      (implies (ok t)
                (and (= (type-of t) (table/list))
                      (seq!seqp (left t))
                      (seq!seqp (right t))
                      (<= (count t) (capacity)))))
[...]
```

>(REDUCE)

```

[...]
```

A.2.3 Executable Functions

The functions `exright` and `exleft` indicate the existence of one or more elements to the right of, or to the left of, the current element respectively. The rewrite rules `exright-spec`

and `exleft-spec` specify the properties of the two functions respectively. The proof of `exright-spec` is more complicated than that of `exleft-spec`. It requires a `split` and an `invoke`.

```

>(typed-function exleft (((t) (table/list))) (bool)
  ((pre (ok t)))
  (> (lp t) 1))
[...]
>(REDUCE)
[...]
>(rule exleft-spec (t)
  (implies (ok t)
    (= (exleft t)
      (> (seq!length (left t)) 1))))
[...]
>(REDUCE)
[...]
>(typed-function exright (((t) (table/list))) (bool)
  ((pre (ok t)))
  (<= (rp t) (capacity)))
[...]
>(REDUCE)
[...]
>(rule exright-spec (t)
  (implies (ok t)
    (= (exright t)
      (not (= (right t) (seq!empty))))))
[...]
>(REDUCE)
[...]
>(SPLIT (> (RP T) 50))
[...]
>(REDUCE)
[...]
>(INVOKE RIGHT-AUX)
[...]
>(REDUCE)
[...]

```

The function `out` indicates whether a current element exists or not. The axiom `out-spec` shows that the left is the empty sequence if there is no current element in `table/list`.

```

>(typed-function out (((t) (table/list))) (bool)
  ((pre (ok t)))
  (= (lp t) 0))
[...]
>(REDUCE)
[...]
>(rule out-spec (t)

```

```

      (implies (ok t)
               (= (out t)
                  (= (left t) (seq!empty))))))
[...]
```

>(REDUCE)

```

[...]
```

>(SPLIT (= (LP T) 0))

```

[...]
```

>(REDUCE)

```

[...]
```

>(INVOKE LEFT-AUX)

```

[...]
```

>(REDUCE)

```

[...]
```

The function `current` returns the current element. It should be called only when a current element exists. This restriction is stated in the pre-condition. The axiom `current-spec` says that the current element is the head of the sequence that is returned by `left`.

```

>(typed-function current (((t) (table/list))) (entry)
  ((pre (and (ok t)
             (not (out t)))))
   (aref (store t) (lp t)))
[...]
```

>(REDUCE)

```

[...]
```

>(INVOKE LEFT-AUX)

```

[...]
```

>(REWRITE)

```

[...]
```

>(rule current-spec (t)
 (implies (and (ok t)
 (not (out t)))
 (= (current t) (seq!head (left t)))))
[...]

>(REDUCE)

```

[...]
```

>(INVOKE LEFT-AUX)

```

[...]
```

>(REDUCE)

```

[...]
```

A.2.4 Procedures

The procedure `init` is the implementation of initialisation for `table/list`. Its post condition is that `table/list` is `ok` and that the the left and right of `table/list` are both the empty sequence. The implementation simply assigns initial values to the pointers. EVES will generate a proof obligation for the procedure whose proof is sufficient to demonstrate the correctness. The proof obligation is proven by the `REDUCE` command.

```

>(procedure init ((pvar (t) (table/list)))
  ((post (and (ok t)
              (= (left t) (seq!empty))
              (= (right t) (seq!empty)))))
  (:= (lp t) 0)
  (:= (rp t) (+ 1 (capacity))))
[...]
```

The procedure `delete` removes the current element from `table/list`. It should be called only when a current element exists. Its post condition specifies that its effect is to remove the head element from the sequence returned by `left`. The implementation decrements the pointer `lp`. The proof requires both an `EQUALITY-SUBSTITUTE` and an `INVOKE`.

```

>(procedure delete ((pvar (t) (table/list)))
  ((initial (t0 t))
   (pre (and (ok t)
             (not (out t))))
   (post (and (ok t)
              (= (right t) (right t0))
              (= (left t) (seq!tail (left t0))))))
  (:= (lp t) (- (lp t) 1)))
[...]
```

```

>(REDUCE)
[...]
```

```

>(EQUALITY-SUBSTITUTE T0)
[...]
```

```

>(INVOKE (LEFT-AUX (STORE T) (LP T)))
[...]
```

```

>(REWRITE)
[...]
```

The procedure `insert` adds an element to the right of the current element, or at the beginning of `table/list` if there is no current element. This new element becomes the current element. Insert should be called only when the count of `table/list` is less than the capacity. Its specification states the effect on `left`, that is, adding the element to `left`.

```

>(procedure insert ((pvar (t) (table/list)) (lvar (d) (entry)))
  ((initial (t0 t))
   (pre (and (ok t)
             (< (count t) (capacity))))
   (post (and (ok t)
              (= (right t) (right t0))
              (= (left t) (seq!tack d (left t0))))))
  (:= (lp t) (+ (lp t) 1))
  (:= (aref (store t) (lp t)) d))
[...]
```

```

>(REDUCE)
[...]
```

The procedure `alter` changes the value of the current element. It should be called only when there is a current element.

```
>(procedure alter ((pvar (t) (table/list)) (lvar (d) (entry)))
  ((initial (t0 t)
    (pre (and (ok t)
              (not (out t))))
    (post (and (ok t)
               (= (right t) (right t0))
               (= (left t) (seq!tack d (seq!tail (left t)))))))
  (:= (aref (store t) (lp t)) d))
[...]
```

```
>(REDUCE)
[...]
```

```
>(INVOKE LEFT-AUX)
[...]
```

```
>(REWRITE)
[...]
```

The procedures `goright` and `goleft` move the current element pointed by the two pointers `lp` and `rp` by one position. They should be called only when such a move is possible, thus the use of `exright` and `exleft` in the pre-condition. Remember that moving the position requires moving an element in the array. The post conditions specify this effect in terms of `left` and `right`.

```
>(procedure goright ((pvar (t) (table/list)))
  ((initial (t0 t)
    (pre (and (ok t)
              (exright t)))
    (post (and (ok t)
               (= (right t) (seq!tail (right t0)))
               (= (left t) (seq!tack (seq!head (right t0)) (left t0))))))
  (:= (lp t) (+ (lp t) 1))
  (:= (aref (store t) (lp t)) (aref (store t) (rp t)))
  (:= (rp t) (+ (rp t) 1)))
[...]
```

```
>(REDUCE)
[...]
```

```
>(EQUALITY-SUBSTITUTE T0)
[...]
```

```
>(REDUCE)
[...]
```

```
>(procedure goleft ((pvar (t) (table/list)))
  ((initial (t0 t)
    (pre (and (ok t)
              (not (out t))))
    (post (and (ok t)
               (= (right t) (seq!tack (seq!head (left t0)) (right t0)))
               (= (left t) (seq!tail (left t0)))))))
[...]
```

```
(:= (rp t) (- (rp t) 1))
(:= (aref (store t) (rp t)) (aref (store t) (lp t)))
(:= (lp t) (- (lp t) 1))
[...]
```

>(REDUCE)

```
[...]
```

>(EQUALITY-SUBSTITUTE TO)

```
[...]
```

>(REDUCE)

```
[...]
```

B EVES versus m-EVES

Jim Grundy reviewed m-EVES in [9]. In this Appendix, we review the main criticisms of m-EVES in [9] and discuss whether improvements have been made in EVES in these areas. EVES preserves those strengths of m-EVES discussed in [9] and makes improvements in many respects.

First we consider those aspects related to m-Verdi.

- *The inability to define recursive procedures in m-Verdi is a severe and unnecessary restriction on the language.* Verdi supports recursive procedures through direct or indirect recursions. This is a major improvement of Verdi over m-EVES.
- *The inability to define stub declarations in m-Verdi is both an inconvenience and a hindrance to the development of a useful system of libraries.* Verdi supports a large range of stub declarations, including function stubs, procedure stubs and type stubs. It is now possible to define a highly abstract declaration in terms of stub declarations and axioms using the stubs. Several models for the highly abstract specifications can then be constructed, by defining the named functions, procedures and types in appropriate forms such as in terms of other theories and concrete implementations. With an added library support facility, a better support for building useful libraries for various usages is now available in EVES. Defining certain entities using stub declarations, we can build more general models, for example an implementation of abstract data type `table/list` whose element type is generic.
- *The language m-Verdi is verbose and cumbersome in appearance, making it somewhat difficult for the user to analyse large predicates.* As mentioned earlier, Verdi has the same weakness since Verdi still uses s-expression syntax similar to the expression syntax of Lisp.
- *The lack of polymorphism in m-Verdi restricts the expressiveness of the language.* Polymorphism is still not directly supported in Verdi. However, with the quite general support of stub declarations in Verdi, parametric polymorphism can be supported through declaring types as type stubs without defining the types in detail. These type stubs may also be used in building different models of specifications.

Without direct support for polymorphism, type checking is more complicated, when it is possible at all. Support for inclusion polymorphism would be difficult within the current framework of Verdi, again if it is possible at all. It is our opinion that, with direct and sufficient support for polymorphism, the language Verdi would be more expressive and reasoning about Verdi expressions would be more efficient. Sufficient support for polymorphism is especially useful for abstract data types and other highly abstract specifications.

We now consider the criticisms of m-NEVER given in [9].

- *There is only a small set of built-in proof commands in m-NEVER, which are quite coarse-grained and can be time-consuming.* The set of proof commands in NEVER extends that of m-NEVER with two types of commands: those for primitive proof steps and those for proof environment management. The commands for primitive proof steps enable a user to use, for example, a single rule or axiom. This is particularly useful when a user intends to transform a formula into a certain form that is more suitable

for a heuristic proof command such as `REDUCE`. The proof environment commands are very useful for accessing different sorts of information related to proofs, especially in a large and complex proof process. Therefore these two sorts of new proof commands in `EVES` improve the proof process. However, the proof commands in `EVES` are still somewhat coarse-grained, as the main proof facility is still based on built-in heuristics; user interaction is mainly for helping the use of the heuristics. Users can do nothing to modify or add to these heuristics.

- *The prover m-NEVER cannot focus on subexpressions of the predicate that a user is trying to prove.* This aspect has been improved in `EVES`, where there are commands supporting a user to focus on subexpressions of the current formula. The use of contextual information in proving a subexpression of a formula is also supported. However, the use of such information is internal to `NEVER`. It would be desirable for this information to be available to the user as well. For example, a window inference interface for `HOL` has been developed in [10] and is known to be an effective technique.
- *m-NEVER's heuristics are good in some areas but weak in others, and can lead to a tendency to specify problems in a manner that suits the theorem prover, rather than the specifier.* Since `NEVER` preserves most of the heuristics employed in `m-NEVER`, `EVES` inherits at least part of this weakness from `m-EVES`. The improvement of `Verdi` over `m-Verdi` in supporting higher level abstraction of specifications lessens its impact. As mentioned earlier, `EVES` supports specifications with stubs and axioms. Different models can then be built for highly abstract specifications. Proofs are needed in building a concrete model for a specification, which can be seen as part of a lower level specification.
- *The simple proof management scheme of m-NEVER makes it hard to use lemmas in proofs.* Here the issue is how well the system supports the declaration and proof of auxiliary results. In `NEVER`, as in `m-NEVER`, there is little support for suspending a proof, in order to state and prove a separate result. It would also be desirable to be able to structure a proof by declaring secondary assertions, then proving them while the main proof is suspended, taking advantage of the current context of the suspended proof. This is not the traditional concept of lemma, but is highly desirable.
- *Reasoning in only one logic is supported in m-NEVER.* Reasoning in `NEVER` is very much the same as that in `m-NEVER`, that is, in only one logic. The difference is that the logic in `NEVER` is an untyped first-order logic, while the logic of `m-NEVER` is a typed first-order logic.
- *A user cannot define her own proof commands (tactics) in m-NEVER.* `NEVER` has a set of new facilities, called *command modifiers*, which can be used to make limited modifications of the proof commands in `NEVER`. Nevertheless, there is still no facility to support user defined proof commands (tactics). With the inclusion of primitive proof commands in `NEVER`, it is natural to program new proof commands based on combinations of existing ones in order to achieve a more flexible and powerful proof system.