Chapter 6

The UNITY Verifier System

In the previous chapters we have demonstrated how the temporal logic and programming notation of UNITY can be extended and exploited so to obtain efficient model checking algorithms. In particular, we have argued that interactive verification based on UNITY logic makes it possible to incorporate design knowledge into the verification process, either by providing or establishing sufficiently strong invariants, or by supplying progress hints in the form of regular expressions. Moreover, the interactive model checking based on UNITY logic takes advantage of the asynchronous computation model of UNITY and of the design information supplied in order to simplify or even eliminate some computations required for verifying properties. In this chapter, we describe how the previously developed theory and the proposed algorithms are implemented as part of the *UNITY Verifier System*, abbreviated as UV System, an interactive symbolic model checker for finite state UNITY programs and propositional UNITY properties.

The design and implementation of an interactive verification system for UNITY (and our extensions of UNITY logic) is motivated by three goals: first, we need to substantiate our claim that the verification of certain concurrent systems can be aided by taking advantage of the simplicity and structure of UNITY logic, and by employing a methodology for making design knowledge a formal ingredient of the verification process; second, we want to provide a useful tool for designers who routinely use UNITY for designing, analyzing, or modeling concurrent systems, which can help them in performing certain tedious and error-prone verification tasks; and third, we strive to construct a system that can serve as a platform for further research on verification techniques and methodologies.

The above goals influenced the overall design of the UV system significantly. In order to demonstrate the feasibility and advantages of the UNITY model checking approach and of the interactive use of it, a system has to be built that can be run on sizable and interesting examples, and that can be compared to other model checker implementations. The desire to build a useful tool calls for an intuitive user interface that allows for easy interactive access to the functionality provided by the model checker, while presenting information about verifier invocations in a manageable way. Furthermore, the intention of using the system as a basis for future research requires its construction in a modular and extensible fashion.

In the remainder of this chapter we describe how these design goals are met by our implementation of the UV System[1]. In section 6.1 we give an overview of the system architecture. Important features of the UV language, in which the programs and properties that are to be submitted to the verifier are written, are discussed in section 6.2. We present some aspects of the user interface of the UV System in section 6.3 and explain how the system is used for interactive program verification. We conclude this chapter with a summary of the implementation and a discussion of extensions to the system that are currently under way or are planned for the near future.

For up-to-date documentation of the UV system and for a more detailed description of various implementation issues, the reader is invited to visit the UV system home page on the world wide web at the following URL:

`http://www.cs.utexas.edu/users/markus/uv2/welcome.html`

[1] The system description is based on the current version, 2.3.3 , of the UV System.

6.1 The UV System Architecture

In order to meet the requirements of modularity, extensibility, performance, and ease of use, the architecture of the UV System has been designed to support the following features:

Object-oriented workspace: entities that are relevant for a verification task, such as programs and properties, are available to the user for direct inspection and manipulation.

Separation of user interface from core functionality: the operations of the verifier are independent from their representation in the user interface; different interfaces can be provided for different tasks or user preferences, experimenting with the interface does not require extensive changes to the system core.

Scriptability: modifying the user interface, performing repeated tasks or series of experiments, or recording statistics in an automated fashion becomes possible without having to recompile the entire system.

Adequate Input Language: a strongly typed language with user definable expressions and data types close to the original UNITY notation makes it possible to conveniently state programs and properties.

Efficient Symbolic Representation: expressions and formulae used to represent various parts of the programs and properties under consideration are encoded and represented by ordered binary decision diagrams (OBDDs, [Bry86]), and can, therefore, often be stored and manipulated very efficiently.

These features are implemented in a system architecture that consists of two separate layers: the *UV kernel* provides the core functionality of the system, while a *GUI* (Graphical User Interface) allows the user to access the kernel in a convenient manner. The architecture is based on the Tcl/Tk system ([Ous94]), which makes it possible to separate kernel and user interface, to make the core functionality of the system available as a set of Tcl commands, and to interactively execute and combine these commands as required by a specific verification methodology or user interface. Moreover, the use of Tcl/Tk as a GUI definition language significantly reduces both code size and development time needed to build a user interface (cf. section 6.4).

The user can interact with the two layers in different ways: the kernel is accessible as the `uvwish` shell, which is an extension of the standard Tcl/Tk windowing shell `wish`. In addition to all the usual wish commands, several UV specific commands have been added to provide the UV functionality. The user can interact directly with the command line interface of the uvwish shell. On the other hand, a GUI is entirely written as Tcl/Tk scripts on top of uvwish. There is a standard GUI provided with the UV system distribution, but it can be modified or replaced by another one without affecting the kernel. Every interaction with the kernel through a GUI can be performed with the uvwish shell alone, although often in a far less convenient way. On the other hand, not all operations of the kernel are necessarily accessible through a GUI. The overall system structure of the UV system is illustrated in figure 6.1. As can be seen the uvwish shell consists of three parts:

The Tcl/Tk Library and Parser provide all the common wish commands and the system initialization, as well as the main event loop for the entire system.

UV Tcl Commands access the special functionality and data representations of the UV system core through a well defined script command interface. Typical commands are uv_init for initializing the UV workspace, and uv_parse for parsing a string as a UV input.

The UV Workspace holds all the data objects generated and manipulated during a verification session such as formulae, programs, and properties. It contains

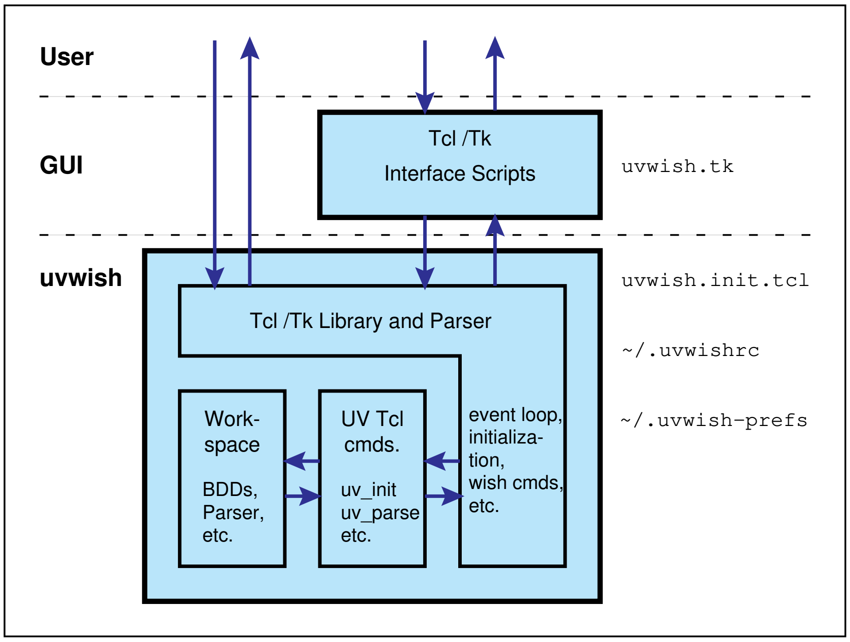

Figure 6.1: The UV System Architecture

an OBDD package for the symbolic representation of boolean functions and many auxiliary data structures. It also implements the actual system operations such as UV parsing and model checking.

While the functionality of the UV system is determined by the UV kernel (consisting of the workspace, the Tcl commands, and the Tcl/Tk interface), its appearance is mostly defined by several Tcl/Tk scripts that define how the UV system is executed:

The loading script `uvwish.init.tcl` is responsible for processing command line options, maintaining the preference file, and starting up the UV kernel.

The preference script `.uvwish-prefs` located in the user's home directory keeps track of user preferences and system settings. It is maintained by the loading script and is not intended to be directly edited by the user.

The interface script `uvwish.tk` defines the graphical user interface. It is executed as last part of the UV system start-up process.

The user script `.uvwishrc` , also located in the user's home directory, is a user-supplied resource script that can contain additional setup or customization commands. These commands are executed after the user interface has been set up.

The UV system obtains its modular and extensible structure mainly from the separation of the user interface from the system kernel, and from the modularization of commands providing the kernel functionality. Moreover, certain implementation-level techniques have been used to keep the overall structure clean and manageable: first, the object-oriented features of C++ as the implementation language are used extensively to ensure the design principles of data encapsulation and separation of functionality (model checking algorithms) from representation (symbolic representation as well as user interface). Second, the parser and the lexical analyzer for the UV input language (cf. section 6.2) are generated from sets of augmented grammar rules, allowing for greater flexibility in making modifications to the input language. Finally the OBDD package and its memory management (including a non-incremental version of a treadmill garbage collector [Bak92]) have been developed as part of the system in order to facilitate a seamless interaction with the other system components while retaining the possibility of modifying the system as part of future research. Meeting the goals of seamless interaction and future extensibility would have become more difficult when using an existing package for symbolic representations.

In the remaining part of this section we discuss briefly two important parts of the UV kernel: the workspace and its implementation in section 6.1.1, as well as the OBDD package in section 6.1.2. Aspects of the GUI and the Tcl interface are presented later in section 6.3, while an overview of the modules of the UV source code can be found in appendix C.

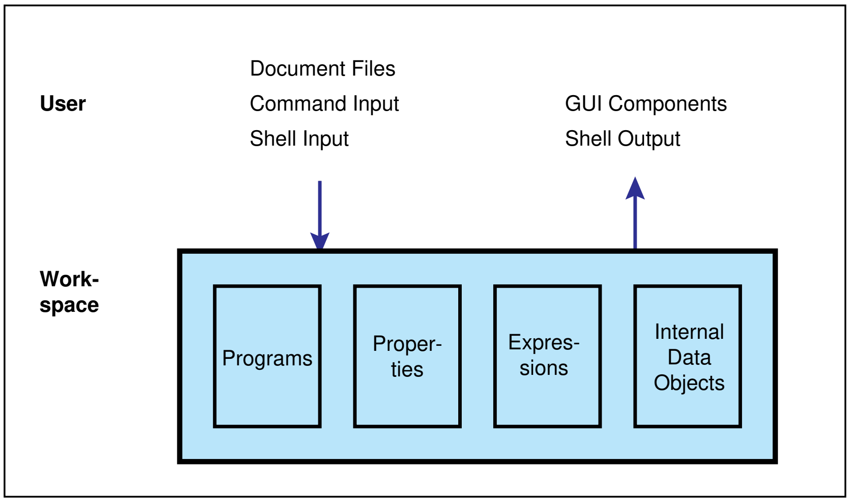

Figure 6.2: The UV Workspace

6.1.1 The UV Workspace

During a verification session the user manipulates objects that are relevant to the verification task at hand. Such objects include UNITY programs, properties, and expressions characterizing relations and sets of program states. These objects (as well as other internally used objects such as symbol tables or status information blocks) are maintained by the UV system in what can be conceptualized as a workspace, a structured collection of relevant objects. An illustration of the workspace structure is shown in figure 6.2.

There are two ways in which objects are introduced into the UV workspace: either they are constructed as a result of a translation from an external textual form according to the UV input language into an internal representation suitable for storage and manipulation, or they are created by performing certain operations with already existing objects. Building a program object from a textual description of a UNITY program is an example of object construction, whereas extracting an expression characterizing a counter-example from a property after a failed model checker invocation is a typical way of creating an object from an already existing one.

For effective and convenient interactive operation of the UV system, the internal representations of the relevant objects in the workspace need to be directly accessible through the user interface. This is accomplished by maintaining external names for those objects as well as mappings that allow the user to access objects by their external names. These external names typically consist of a prefix that characterizes the kind of an object (for instance `#expr` for expressions) followed by a unique number for that category. Certain objects can also be referred to by user-provided names.

The most important entities a user deals with during a verification session are the following:

documents and user input externally represent workspace objects such as programs and properties,

programs are the system models to be analyzed and verified,

properties are specifications to be checked for certain programs,

expressions are, in general, formulae over the state space of certain programs characterizing sets of states, relations, invariants or parts of properties,

invariants are expressions that play a special role in the verification of UNITY programs.

regular expressions are formal hints used in checking progress properties.

In the following we briefly characterize these entities and describe typical operations performed with them.

Documents and User Input

In order to introduce UNITY programs, properties, and expressions into the UV system the user provides an ASCII text description of these objects. The syntax of this external description is governed by the UV language. Input strings can be read from document files or can be provided interactively through some user input window. Document files typically contain descriptions of programs and properties to be checked, whereas interactive input is mostly used to evaluate interesting expressions or to experiment with additional properties.

Each input string provided by the user is processed by the UV Parser, which performs syntax and type checking and generates an internal representation for correctly described objects. In case syntax or type errors are encountered, a suitable error message is returned and the location of the error in the input string is reported. Parsing and introducing new objects into the workspace is done incrementally as newly described objects are added to the workspace without altering or removing existing ones.

Programs

Programs define the models (state transition systems) for which model checking is to be performed. In particular, any program defines a state space of all possible values of its variables, a set of initial states, and a labeled transition relation determining which state can be reached by executing a program statement in a given state. Furthermore there are invariants associated with every program, corresponding to sets of states that are closed under the execution of the program.

A successfully parsed program is entered into the workspace as part of the *program table* containing all currently defined programs. The user can display status information of all programs contained in the program table, and can invoke operations on the programs such as computing the strongest invariant.

Properties

Properties are the specification formulae that need to be checked for programs. Each property is associated with one program, of which it expresses a certain behavior.

A successfully parsed property is entered into the workspace as part of the *property table* containing all currently defined properties. The user can display status information of all properties contained in the property table, can invoke operations on the properties, such as model checking the associated program for the given property, or can remove properties from the workspace.

Expressions

Expressions represent typed values over some state space, typically the state space of a particular program or the global state space. Boolean expressions over the state space of a program often denote sets of program states; similarly, boolean expressions over the Cartesian product of a program state space with itself denote transition relations.

Expressions are introduced into the workspace either directly through parser invocation on some user input, or as a result of certain operations on other expressions, programs, or properties. A successfully parsed or created expression is entered into the workspace as part of the *expression table* which contains all currently defined expressions. The user can display information about expressions contained in the expression table, and can, furthermore, refer to them by using external expression names when parsing subsequent input.

Invariants

Invariants are boolean expressions over the state space of a program that play an important role in the model checking of properties of that program. Finding suitably strong invariants is critical to efficient interactive model checking of UNITY programs.

The UV system maintains at most three invariants for each program: the *type invariant* expressing that every program variable takes on values only from the domain determined by its type, the *current invariant* which is the conjunction of all invariants established for the program during a verification session, and, optionally, the *strongest invariant* characterizing the set of reachable states of the program.

Regular Expressions

Regular expressions are part of generalized progress properties introduced in chapter 4. They play an important role in utilizing action-based design knowledge in the model checking of progress properties.

Currently, the UV system maintains regular expressions only as part of generalized progress properties. Additional operations, such as keeping regular expressions as separate objects and performing algebraic manipulations on them will be provided in future system revisions.

6.1.2 Symbolic Representation Using OBDDs

Ordered binary decision diagrams are long known to be an efficient symbolic representation for boolean functions encoding sets of states and transition relations (cf. 2.3.2). We chose OBDDs as symbolic representation mostly because of the success of BDD based verification methods and implementations. It is, however, important to note that the model checking procedures for UNITY, presented in chapters 3 and 5, are independent of any particular symbolic or even explicit representation. For us, OBDDs make it possible to verify a wide range of programs, but it might still be the case that other representations (such as explicit state enumeration, or certain representations based on predicate calculus for infinite state systems) are better suited for some classes of programs.

Ideally, it should be possible to keep a strong separation of the symbolic representation and the actual verifier operations. That makes it possible to incorporate improved representations, or even to use multiple representations for different verification tasks or inputs. When work on the UV system was begun in 1992, an OBDD package was developed as a separate C++ class with the goal of supporting this separation by accessing the symbolic representation from the model checker and parser only through a well defined interface (cf. appendix C). Although by now there are complete OBDD packages and libraries available (for instance, David Long's OBDD package developed at CMU), we continued to use our own OBDD package for two main reasons: first, the intimate understanding of the package allows us to experiment with different ideas and implementations; second, as part of a one-person prototype development it is more important to concentrate on the algorithmic contributions than to attempt to incorporate all new ideas for the improvement of symbolic representations. Nonetheless, it was important to provide an efficient implementation of the low-level symbolic representation structures and manipulation operations documented in the literature. The techniques and methods currently implemented include the use of reduced OBDDs as described in [BBR90], of a combined *and-exists* operation in computing relational products ([McM93], of quantification ordering in synchronous transitions similar to those described in [BCM91], of restriction [CM90], and generalized cofactoring [TSL+90].

In addition to provisions for taking advantage of the monotonicity of predicate transformers in early termination of fixpoint computations, the current implementation also uses a special second level cache for memoizing certain and-exists computations in addition to the standard *if-then-else* (ITE) cache ([BBR90]. We observed, in particular, that in the presence of relatively small deterministic multiple assignments, typical of many UNITY programs, a hit-rate of above 30% was achieved for the and-exists cache, resulting in a significant gain in performance. The and-exists caching technique has been used in other systems as well [Fil94], where somewhat smaller performance improvements on large examples have been observed.

6.2 The UV Input Language

A user of the UV system describes most entities that play a role in a verification session by using the UV input language. UNITY programs, UNITY properties, and expressions involving program variables and predicates are typical entities, which are entered into the UV workspace after being parsed and compiled into an internal OBDD based form suitable for further processing. Therefore, the UV input language defines the set of possible user inputs, and provides the user with a convenient way of expressing programs, properties, and expressions. These goals are met by designing the UV input language to be close to the original UNITY notation while having the following features:

Finiteness: in order to ensure finiteness of all state spaces, the UV input language allows only finite data types.

Strong Typing: a strong type system makes it possible to derive information about invariants from the program text.

Complete Set of Properties: all properties of UNITY logic are implemented including the generalized progress properties introduced in chapter 4.

Statement Labels: in extension to the traditional UNITY notation, all statements have unique labels by which they can be identified in debugging and formalizing progress hints.

In the following, we briefly describe some aspects of the UV input language. In particular, we discuss parse units, types and type checking, expressions, programs, and properties. In our presentation we show some rules of the grammar for the UV input language. A complete formal description of the grammar for the UV input language and of the UV type system can be found in appendix A.

6.2.1 Name Space and Scoping

The UV system has two name spaces: one for variable, constant, type, program, and field names (henceforth referred to as object names), the other for action names. The structures of the name spaces determine the visibility of names in a system description: associated with each action and each object is a *scope*, defined as parts of the description in the UV input language from which the action or object can be referred to by its name. Both action and object name space have a tree structure with the global scope as their roots and sub-scopes as their children. Sub-scopes for both objects and actions are introduced with every program definition, additional sub-scopes for objects (field names) are created for each record type.

Every object and action is visible in the scope where it is declared and recursively in all children of that scope, subject to the restriction that names in sub-scopes (lower scopes) hide the same names in outer scopes. Names declared in the same scope of the same name space have to be unique, e.g. , it is illegal to declare two enumeration types with a shared enumeration constant name in the same scope, since the enumeration constants are declared in the same scope.

The action name space has an empty global scope and a local scope for each program containing exactly the statement labels of that program.

The object name space has a global scope containing certain predefined names, such as the type `boolean` and the constants `true` and `false`, as well as all object names declared by the user globally (e.g. , program names are globally declared constants of program type). On the other hand, object names declared inside programs (or field names of record types) have local scope, i.e., are only visible from within the program, or from within expressions and properties that are placed into the program context.

6.2.2 Types

The benefits of using type systems in programming are well known. In addition to the ability of recognizing certain program errors at compile-time, a suitable type system provides the user with invariants that can be derived from the program text by simple syntactic operations. These invariants come at no cost for the programmer since there are no proof obligations to establish them.

We, therefore, adopt a simple strong type system of finite types for the UV input language. In this type system every expression occurring in a program or in a property has a statically well defined type. In the following we describe the available types together with the operations on them, and sketch the rules for correctly typing programs and properties.

The UV Type System

The type system of the UV input language consists of nine different kinds of types. The simple types are `boolean`, `number`, `cyclic`, `bits`, `int` (finite range integers), `enum` (enumeration), and `program`; the structured types are record and mapping. The simple types are all finite, and the structured types are finite types constructed from simpler finite types.

Among the simple types, the number and program types are restricted in the sense that the user cannot declare variables of these types: any program can be thought of as the definition of a constant (the program name) of type program, and any expression made up from only number literals and arithmetic operators is given a number type. There are no operations defined on the program type, but program constants are used in specifying the context of properties.

The boolean type has two predefined constants `true` and `false` and the usual boolean operators. Finite enumeration types are given by ordered list of unique enumeration constants; these constants define the possible values of the type as well as a linear order (from smallest to greatest element) on it. Additionally, the

UV type system provides three finite-range integer types, that differ slightly in the way they relate to the actual set of integers, i.e., in the way the arithmetic operators `+` and `-` are defined (the comparison relations are similarly defined for all integer types, as the relation on the corresponding integers):

Cyclic types: `cyclic(n)` represents the integers modulo the positive natural `n`, i.e., the cyclic group of `n` elements. Inequality relations are to be used carefully, since they do not obey monotonicity laws (e.g. , in `cyclic(5)`, `2`$<$`4` but not `2+2`$<$`4+2`).

Bits types: `bits(n)` represents `n`-bit numbers for which arithmetic is performed modulo 2^{n}. They are similar to the `cyclic` types, but support additional bit-level operations[2].

Interval types: `int(m..n)`: represents the intervals of integers from `m` up to and including `n`. The type of an expression involving interval type variables is the syntactic minimal interval type suitable for the result; for instance, if `x` has type `int(2..7)`, the expression `x-3` has type `int(-1..4)`. No modulo arithmetic is performed.

The two kinds of structured types are the record type and the mapping type, the latter of which can be thought of as a generalization of array types. A record type is defined by a non-empty set of pairs of *field-names* and previously defined types. A mapping type is determined by two previously defined types, an index type and an element type. A variable of a mapping type is a (finite) mapping from the index type to the element type. Arrays in languages like C++ or PASCAL are special cases of mappings in which the index type is some integer or range type. A few examples of type declarations are:

```
type Range = int(1..8);
```

[2]not implemented currently

```
type State = enum{idle, trying, critical};
type Process = {id: cyclic(16); state: State};
var process: Process;
var network: Range -> Process;
type Monitor = Process -> enum{never, once, often};
```

Note that the index type of a mapping type can be a structured type (see `Monitor` in the example above). Accessing components of a record type or elements of an index type is expressed with the function application operator `.` as in `process.state` or `network.(i+1)` .

Subtypes

The notions of *subtypes* and *type coercion* play an important role in determining whether a given program, property, or expression is typed correctly: only correctly typed input is compiled into the internal representation, and any (sub-)expression of a correctly typed input has a unique, statically determined type.

Informally, a type S is a subtype of type T if every element of S can be regarded as an element of T. For instance, extending a record type R by a new component yields a subtype of R, since any element of the new type can be regarded as an element of R (in which the value of the new component is ignored).

The subtype relation on integer types is, at the first glance, somewhat unusual: `number` types are subtypes of all other integer types, `cyclic` and `bits` types are only subtypes of themselves, and `int` types are subtypes of any `int` type. Types `cyclic` and `bits` types are kept incompatible with other integer types because they have a different algebraic structure. Compatibility of arbitrary `int` types was chosen in order to support the informal semantics of such types as providing a *window* into the infinite set of integers. In particular, this compatibility is a requirement for being able to handle assignments of the form `x := x+1` where `x` has interval type `int(m..n)`; the right-hand side has type `int(m+1..n+1)` which needs to be a subtype of `int(m..n)` in order to comply with the type rule for assignments. Instead of disallowing such an assignment altogether, it is given the semantics that the value of the right-hand side is some arbitrary value in the target range from `m` to `n` in case the value of `x+1` falls outside this range. An implementation is free to choose that value arbitrarily.

The rationale behind this semantics is the following: an assignment such as the above is inappropriate for a finite state program, since it can generate an infinite program behavior. The only reasonable occurrence of such an assignment should be guarded by some predicate `b` that effectively restricts the growth of the value of `x`. The intention should be that in an actual program execution the statement is never enabled in a state in which the right-hand side produces an out-of-range value. In other words, by including an assignment such as `x := x + 1 if b` in a program, the user actually generates the proof obligation `invariant b ==> x<n`, which should be checked as a required property of the program.

The subtyping relation for structured types is defined recursively as follows: a record type `R` is a subtype of a record type `S`, if and only if every field-name of `S` occurs in `R` and for each field-name `f` of `S` the type associated with `f` in `R` is a subtype of the type associated with `f` in `S`. A mapping type `M` defined as `A->B` is a subtype of a mapping type `N` defined as `C->D`, if and only if `B` is a subtype of `D` and `C` is a subtype of `A`.

Type Checking of Expressions

An expression is well-typed if the UV type checking algorithm successfully labels all subexpressions with their types. The type checking algorithm proceeds by labeling subexpressions with their types in a bottom-up fashion starting with literals and variables, while requiring that the types of any subexpressions occurring as arguments to some operator be compatible with that operator.

6.2.3 Parse Units

The input to the UV parser (more precisely, to the `uv_parse` command of the `uvwish` shell) consists of a sequence of input units, each of which either declares some types or variables, or describes a program or a property, or defines an expression.

All units that have been parsed correctly and have passed the type checking successfully are compiled into an internal OBDD-based representation. In particular, programs are internally represented as a set of statements, each of which is symbolically represented by a boolean function characterizing the transition relation of the statement. Similarly, the internal representation of a property includes the OBDDs representing the expressions from which the property is built, and – in the case of generalized progress properties– also a representation of the regular expression that is part of the property. Expressions are compiled into vectors of OBDDs, the size of which is determined statically by the type of the corresponding expression.

Programs

A program consists of a program name and four sections: the `declare`-section declares local types and variables and, thereby, defines the local state space of the program; the `always`-section defines transparent variables ([CM88]) that serve mostly as abbreviations to render the program easier to read and understand; the `initially`-section defines the set of initial states of the program, and the `assign`-section defines the state transition in terms of asynchronous program actions. Several examples of programs are given in chapter 7.

A program is well-typed if the following conditions are met: all of its expressions are well-typed, the types of all expressions in the *always* section are subtypes of the declared types of the corresponding transparent variables, all expressions in the *initially* section are of type boolean, all expressions appearing as guards in statements in the *assign* section are of type boolean, and the types of all expressions appearing on the right-hand side of the assignment operator are subtypes of the corresponding types of the left-hand side variables, record components, or mapping components.

Properties and Expressions

A property is defined in the context of exactly one program, which is named in a context declaration preceding the property. As a consequence, all global symbols and all local symbols defined in the associated program can be referred to in the property. Some examples of properties of a program named `sample` are:

```
in sample: n=3 unless m>0;
in sample: true --> n=3 by [alpha]*[beta];
```

A property is well-typed if and only if all of its expressions are well-typed, and – unless the property is a **constant** property – are of type boolean.

6.3 The User Interface

Interaction with the UV system can take place at two different levels: the Tcl interface provides access to the core functionality of the UV workspace, whereas a GUI makes some of that functionality accessible in a more convenient way. In the following, we describe these two interfaces in some more detail: first, we describe the *standard* GUI that is provided with the UV system distribution; then, we briefly summarize the underlying Tcl interface.

6.3.1 The Standard Graphical User Interfaces

When interacting with the UV system through its graphical interface the user typically deals with the following parts:

The Command Window gives access to system commands, displays system messages, and allows the user to invoke the parser.

Document Windows are created for each document file in use and allow the user to edit, save and parse documents.

The Program Table Window lists all current programs and allows the user to examine and perform operations on them.

The Property Table Window lists all current properties and allows the user to examine and perform operations on them.

The Expression Table Window lists information about all current expressions.

Property Information Windows display detailed status information about selected properties.

In the following we show typical instances of these interface components and briefly explain how they can be used.

The Command Window

The command window consists of the following four parts arranged from top to bottom; see figure 6.3 for an example.

The Menu Bar provides access to various system operations.

Command Buttons invoke operations on the command window.

The System Message Area displays system messages such as parser error messages or status messages.

The User Input Area allows the user to interactively parse and process UV input.

The menu bar contains a pull-down menu labeled `File` providing operations for creating new documents (`New`), opening an already existing document (`Open...`), and for terminating a verification session (`Quit`). Other menus for accessing system

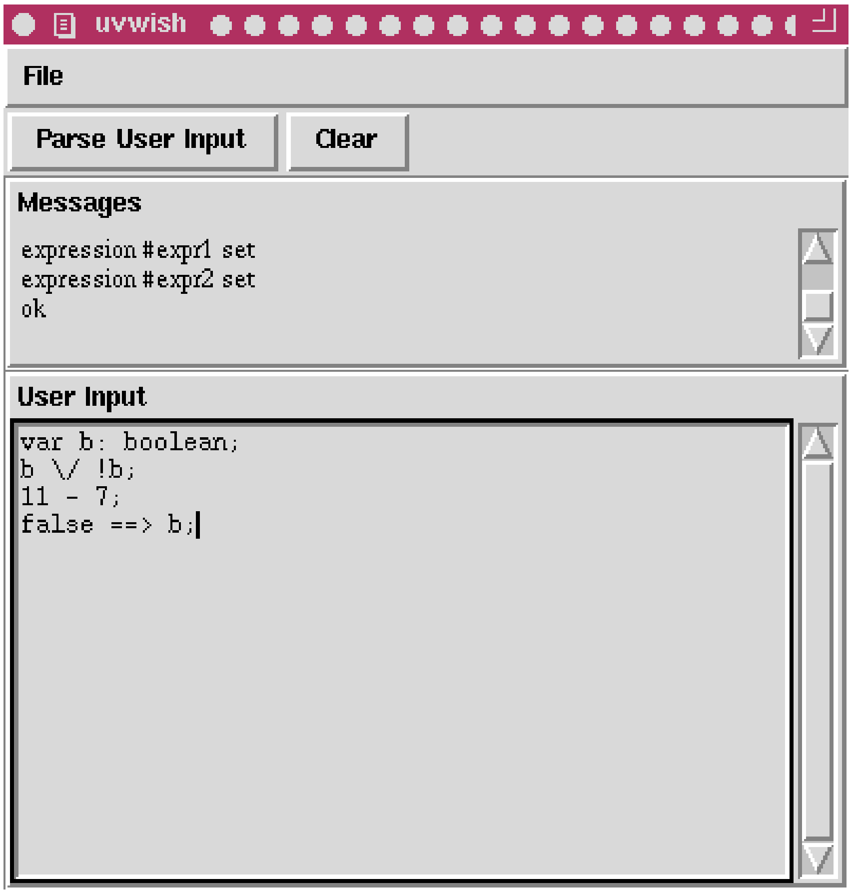

Figure 6.3: A Command Window

level information, setting preferences, or invoking tools like formulae browsers will be added in future releases.

There are two command buttons, one labeled `Parse User Input` for running the UV parser on the entire content of the user input area of the command window, the other labeled `Clear` for clearing both the message area and the user input area of the command window. Status and error messages produced by the parser are displayed in the message area of the command window, and the error position is indicated in the user input area by highlighting the token that caused the error.

Document Windows

A document window is used for viewing and editing input to the UV system, and for filing and parsing operations on such input documents. A document window consists of the following three parts listed as they appear in the layout of the window:

Command Buttons allow the user to perform filing and parsing operations on the document.

The Document Message Area displays messages that result from performing operations on the document.

The Document Content Area is an editable text area that can be used for browsing, writing, or modifying the document content.

An example of a typical document window is shown in figure 6.4.

A document window contains four command buttons that cause the following operations to be performed:

`Parse` runs the UV parser on the entire document content of the document window. Status and error messages produced by the parser are displayed in the message area of the document window, and the error position is indicated in the document content area by highlighting of the token that caused the error.

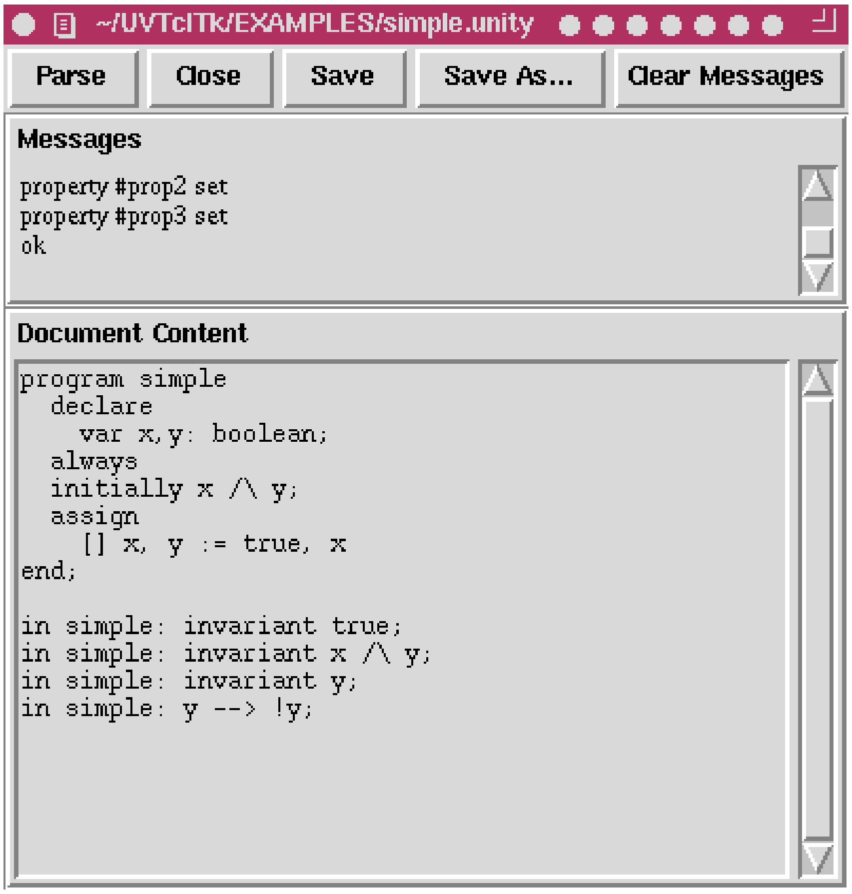

Figure 6.4: A Document Window

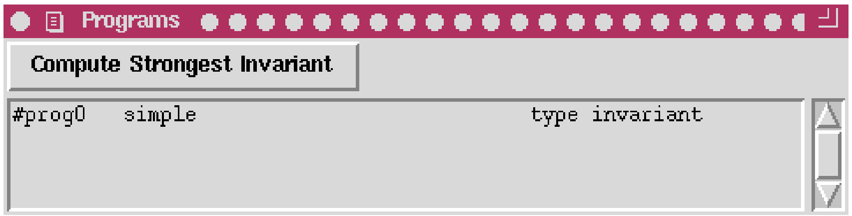

Figure 6.5: A Program Table Window

`Close` closes the document window. Changes made to the document content that have not been saved are lost.

`Save` saves the document to a file. If the document was opened from a file, changes are saved to that file. If the document was newly created, the user is prompted for a new filename under which the document is to be saved.

`Save As...` saves the document to a file, but always prompts the user for a filename under which the document is to be saved.

`Clear Messages` clears the message area of the document window.

The Program Table Window

All programs currently in use are listed in the program table window. The window consists of a command button labeled `Compute Strongest Invariant` and a scrollable list of all programs currently in the UV workspace. An example of a typical program table window is shown in figure 6.5.

For each program one line is displayed containing the program ID, the program name, and information about which invariant has been computed for the program so far. Program lines can be selected with the usual click and drag techniques: clicking the command button causes the strongest invariant to be computed for all selected programs. Although the UV kernel implements different algorithms for computing the strongest invariant (such as forward chaining and iterative squaring), the command button invokes the algorithm that has been found to be the most

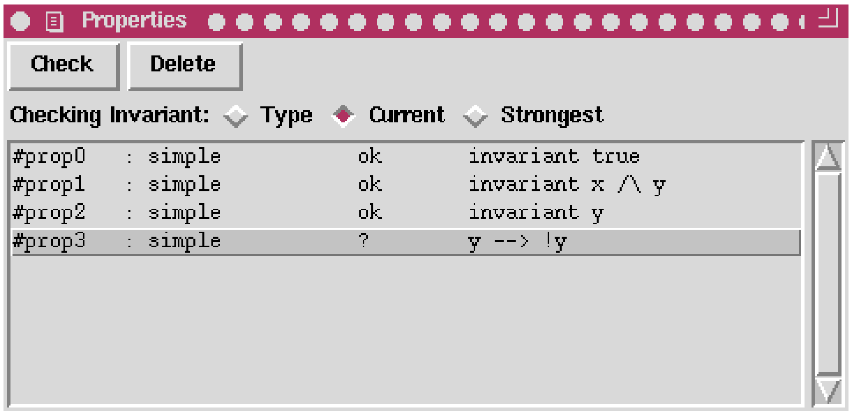

Figure 6.6: A Property Table Window

efficient for most programs. This algorithm is called *frontier forward chaining*; its name is due to the fact that in successive iterations of the forward exploration of the state space successors of those states are determined that have been added only in the previous iteration.

The Property Table Window

All properties currently in use are listed in the property table window. The window consists of the following three parts given in the order in which they appear in the layout of the window:

Command Buttons allow the user to perform operations on selected properties.

Invariant Radio Buttons are used to specify which invariant is to be used in checking properties.

The List of Properties is a scrollable list of all properties currently in the UV workspace.

An example of a typical property table window is shown in figure 6.6.

A property table window contains two command buttons: the first, labeled `Check`, invokes the model checker on all currently selected properties; the second, labeled `Delete`, removes the selected properties from the workspace.

The radio buttons are used to select whether a model checker invocation should use the type invariant, the current invariant, or the strongest invariant of the respective programs for which properties are checked.

For each property in the list of properties, one line is displayed containing summary information about the property, namely the property ID, the name of the program associated with the property, the checking status of the property, and the beginning of the textual representation of the property. The checking status is one of the following four:

`new` indicates that no checking attempt has been made for the property.

`?` indicates that the property has not yet been proved to hold in the associated program, but that there might be a suitably strong invariant with respect to which the property could still be proved.

`ok` indicates that the property has been proved to hold in the associated program.

`fail` indicates that the property has been proved not to hold in the associated program by checking it with respect to the strongest invariant of the program.

Property lines are selected for subsequent operations with the usual click and drag techniques. Double-clicking on a property line with the first mouse button brings up a property information window for this property containing more detailed information about the checking status.

The Expression Table Window

All expressions currently in use are listed in the expression table window. The window consists of a list of entries with one line for each expression. An example of an expression table window is shown in figure 6.7.

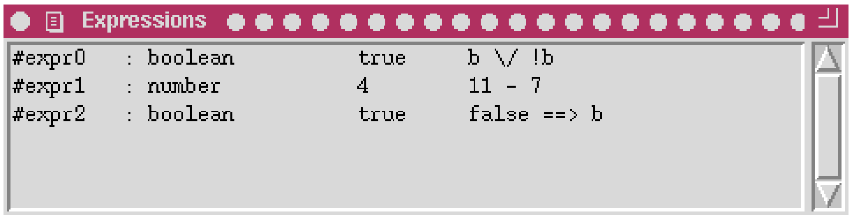

Figure 6.7: An Expression Table Window

For each expression in the UV workspace, one line is displayed characterizing the expression. Each such line consists of the expression ID, the type of the expression, the value of the expression (if the expression represents a constant, otherwise a number in square brackets showing the number of OBDD nodes required in the symbolic representation of the boolean vector encoding of the expression), and the beginning of the textual representation of the expression.

Property Information Windows

For each property listed in the property table, window a property information window can be displayed by double-clicking on the entry of the property in the property table list.

The property information window displays more detailed information about the property and is useful for analyzing model checker output and for debugging properties and their associated programs. The information presented in a property information window includes the name of the program associated with the property, a textual representation of the property, and a list of status items, each providing information about a certain aspect of the checking status of the property. An example of a property information window is shown in the figure 6.8.

Each status item is labeled with an identifying category. Some status items have additional information associated with them in the form of some expression. If such an expression is present, a button labeled `Add Expr` appears at the right of the corresponding status item line. Pressing that button causes the expression to

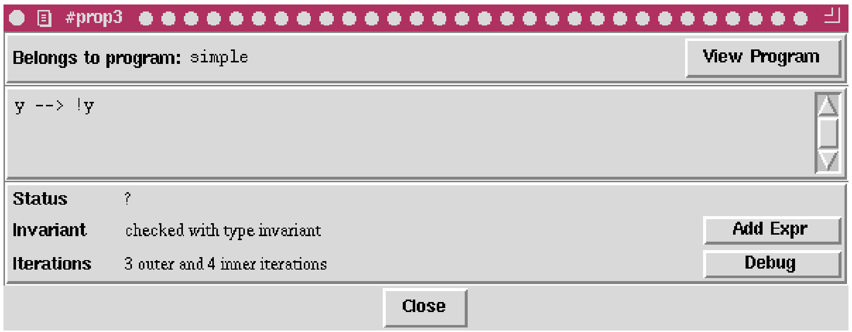

Figure 6.8: A Property Information Window

be added to the expression table window, thereby being made available for further investigation.

The following table lists all status items by their categories and describes the information available through them:

`Status` shows the checking status of the property as displayed in the property table window.

`Invariant` identifies the kind of invariant used for checking the property. This invariant is made available by pressing the `Add Expr` button.

`Implication` states the result of the implication check required for **invariant** and **co** properties. A predicate characterizing all states (within the checking invariant) falsifying the implication is available by pressing the `Add Expr` button.

`Safety` states the result of the safety check required for **co** , **unless** , **stable** , **invariant** , **constant** , and **ensures** properties. In particular, it names a program statement violating the condition (if there is any). If there is such a violating statement then a predicate characterizing all states (within the checking invariant) from which an execution of named statement violates the safety condition is available by pressing the `Add Expr` button.

`Transition` states the result of the helpful transition check required for **transient** and **ensures** properties. In particular, it names a helpful program statement (if there is any).

`Value` shows a value for the expression of a **constant** property for which the stability condition is violated (if there is such a value). If there is a value that is changed by some statement, then such a value is made available by pressing the `Add Expr` button.

`Iterations` presents the number of outer (least) and inner (greatest) fixpoint iterations in the checking of leads-to and generalized leads-to properties. A predicate characterizing all states (within the checking invariant) for which there is a fair program execution not satisfying the leads-to property is made available by pressing the `Add Expr` button.

Furthermore, all status items characterizing counter-examples (i.e., the items labeled `Implication`, `Safety`, `Value`, and `Iterations`) have another button attached to them, which is labeled `Debug`. Pressing this button brings up a new window with a witness for the violation characterized by the corresponding status item. Pressing the `Close` button closes the property information window.

6.3.2 The Tcl Interface

The Tcl interface of the UV system consists of a set of commands and some special Tcl variables added to the standard windowing shell `wish`. The entire functionality is available through these commands and variables: in particular, the standard GUI presented in the previous section is implemented completely as a set of Tcl/Tk scripts.

The commands specific to the UV system as part of the `uvwish` shell are listed briefly in the following table; a detailed description of these commands can be found in appendix B:

`uv_check` invokes the model checker.

`uv_expr` accesses information about expressions.

`uv_info` accesses information about the UV system.

`uv_init` initializes the UV system.

`uv_option` sets and displays system options and parameters.

`uv_parse` parses a string of the UV language and adds the defined entities to the UV workspace.

`uv_prog` accesses information about programs.

`uv_prop` accesses information about properties.

`uv_si` computes strongest invariants.

Currently, there are three Tcl variables that can be used to control the behavior of some UV commands, or communicate information between executing UV commands and other active Tcl/Tk scripts:

`uvAbort` is a boolean variable that when set to 1 causes the potentially time-consuming UV commands `uv_check` and `uv_si` to be aborted.

`uvProgress` is an integer variable that counts the number of major iterations during `uv_check` invocations for leads-to properties and during `uv_si` invocations.

`uvSubProgress` is an integer variable that is set to the percentage of statements checked in `uv_check` invocations for safety and basic progress properties, and in each major iteration of a `uv_check` invocation for a leads-to property.

These variables are used in the standard GUI, for instance, in order to display computation progress bars indicating the status of a lengthy UV command execution (such as computing a strongest invariant or checking a property of a complex program). This is possible because the potentially time-consuming commands `uv_check` and `uv_si` periodically process pending events, thereby updating the user interface display and checking the variables mentioned above.

6.4 Implementation Summary and Extensions

The UV system in its current form is the result of a research and implementation effort that began in 1992 as an experimental OBDD based model checker for UNITY programs, which was written in Scheme [RC86], and ran on a Macintosh computer. Since then, many significant improvements to the symbolic representation have been made, the system has been written in C++ under UNIX and the X Windows system, and the theory of incorporating design knowledge has been developed. The first version of the UV system was made publicly available in December 1994 [Kal94, Kal95b]. After a redesign and clean-up of the system architecture that had grown over a period of two years, the second and current version of the UV system was released in October 1995. Besides the new structure and the interface based on the Tcl/Tk package, version 2 improved the efficiency of the symbolic representation and the expressiveness of the UV input language. We expect the UV system to continue to be a platform for conducting and evaluating further research in verification of concurrent programs in the near future.

The current version, 2.3.3 , of the UV system is based on Tcl 7.4 and Tk 4.0, and has been successfully tested on SunOS 4.1.3 and Solaris 2.4. It was developed using the GNU suite of development tools, including the parser generator *bison* and the lexical scanner generator *flex*. The UV system source consists of about 36,000 lines of commented code. About 24,500 lines are written in C++ , 9,000 lines are C++ code generated by *bison*, *flex* and the *genclass* type generator, and the remaining 2,500 lines are Tcl/Tk code defining the user interface. Overall, 32% of the code deals with parsing and compilation of UV input into the internal OBDD representation, 26% is needed to implement the symbolic OBDD representation and the symbol table handling, about 23% is used on the model checking algorithms and related data structures, 12% is spent on implementing the Tcl interface, and only about 7% of the code makes up the standard GUI. It is worth mentioning that this confirms one of the greatest alleged advantages of using a scripting language like Tcl/Tk for building user interfaces, namely that the code size and the time required to build, modify, and maintain such an interface is significantly lower than traditional user interface designs, in which the interface code is typically written in C++ using some interface toolkit. In the first version of the UV system, we had written an interface based on the Motif[Hel92] toolkit, that not only took up more than 30% of the entire code, but also was more difficult to modify and adjust to the changing requirements of a system that is used as an experimental platform for research.

In spite of its current size and its established usefulness, there are some implementation aspects of the UV system that need to be improved. They can be divided into the following three groups:

Symbolic Representation: several well established techniques for improving the OBDD performance need to be implemented, such as dynamic variable reordering [Rud93] and partitioned OBDD representations [HD93, Jai96];

UV Language: quantified statements and formulae need to be introduced in order to improve the ease of expression of the input language; support for regular expression hints as separate objects and for performing algebraic operations on them could simplify working with progress properties;

User Interface: a better presentation of debugging information in the form of counter-example traces, and the addition of formula browsers could facilitate the user's task of finding and debugging program or specification errors.

In addition to these implementation-oriented extensions, future research concerning compositional reasoning will prompt the need for support of advanced structuring and reasoning methods, which are expected to utilize the modular and extensible structure of the current version of the UV system.