Appendix A

The UV Input Language

In this appendix we present the complete grammar of the UV input language of version 2.3.3 of the UV system. First we introduce our grammar notation in section A.1, and describe lexical conventions in section A.2. Then we show the grammar rules in section A.3. We conclude this appendix in section A.4 with a brief presentation of language extensions that will be implemented in future revisions of the UV system.

A.1 Grammar Notation

The grammar for the UV input language is presented in an extended Backus-Naur Form (EBNF). There are three EBNF *meta symbols*, namely ::= for separating left and right-hand side of grammar rules, | for separating alternative right-hand sides of a grammar rule, and ( )* to denote any finite repetition of the items enclosed in parentheses.

Non-terminals of the grammar are represented as strings enclosed in angle brackets, as in <*name*>, while *terminals* are represented using typewriter font, as in `initially` or `=!=`.

A.2 Lexical Conventions

The UV input language is case sensitive. The input is made up from a sequence of tokens and whitespace (i.e., blanks, tabs, and newline characters as well as comments). Whitespace is significant only for separating tokens, beyond that it is ignored.

Tokens are either terminals as appearing in the grammar rules below, or strings of characters representing the <*number*> , the <*name*> , or the <*external*> non-terminals.

A comment starts with the characters "//" (without the quotation marks) and ends with the first subsequent newline character.

A number as generated by the non-terminal <*number*> is a string of one or more digits, restricted by some implementation dependent limitations on the number size. It is guaranteed that 15 bit numbers (i.e., numbers from 0 to 32767) are supported.

A name as generated by the non-terminal <*name*> is any string of one or more characters made up from letters and digits starting with a letter. Names are case-sensitive and the following keywords of the UV language are not available for names: `assign`, `by`, `co`, `const`, `constant`, `declare`, `end`, `ensures`, `if`, `in`, `initially`, `invariant`, `program`, `stable`, `transient`, `type`, `unless`, and `var`.

Names are used for denoting variables, programs, types, statement labels, and record fields. Examples of names are `hello`, `isDone`, and `t34y0`.

An external name as generated by the non-terminal <*external*> consists of the string `#expr` followed (without any whitespace in between) by a string of one or more digits. The maximum number of allowed digits is implementation dependent, but at least four digits are guaranteed. External names are used for denoting expressions that have been generated interactively during a session with the UV system. Examples of external names are `#expr76` and `#expr003`.

For ease of reference we list in the following the remaining tokens of the UV input language: `:=`, `(`, `)`, `[`, `]`, `;`, `:`, `,`, `..`, and `-->`.

A.3 Grammar Rules

The following presentation is divided by the syntactic categories of the grammar.

A.3.1 Expressions

Expressions are generated by the non-terminal <*expr*> . The following table lists all operators in increasing order of their binding power (operators in the same group share the same binding power, in which case the operators associate to the left – if allowed by the typing rules):

`==` (boolean equivalence), `=!=` (boolean antiequivalence, exclusive or)
`==>` (boolean implication), `<==` (boolean follows-from)
`/\` (boolean conjunction), `\/` (boolean disjunction)
`!` (boolean negation)
`=` `!=` `>` `>=` `<` `<=` (nonboolean relational operators)
`+` (addition), `-` (subtraction)
`+` (unary plus), `-` (unary negation)
`.` (record field access, mapping indexing)

<*expr*> ::= <*name*>
| <*external*>
| <*number*>
| `(` <*expr*> `)`
| <*expr*> `.` <*name*>
| <*expr*> `.` <*expr*>
| `+` <*expr*>
| `-` <*expr*>
| <*expr*> `+` <*expr*>
| <*expr*> `-` <*expr*>
| <*expr*> `=` <*expr*>

| <expr> != <expr>
| <expr> > <expr>
| <expr> < <expr>
| <expr> >= <expr>
| <expr> <= <expr>
| true
| false
| ! <expr>
| <expr> /\ <expr>
| <expr> \/ <expr>
| <expr> ==> <expr>
| <expr> <== <expr>
| <expr> == <expr>
| <expr> =!= <expr>

<exprList> ::= <expr> (, <expr>)*

A.3.2 Types

<type> ::= <name>
| boolean
| int(<expr> .. <expr>)
| cyclic(<expr>)
| bits(<expr>)
| enum(<nameList>)
| { <compList> }
| <type> -> <type>

<nameList> ::= <name> (, <name>)*

$<compList>$ ::= $<compItem>$ (, $<compItem>$)*

$<compItem>$ ::= $<name>$: $<type>$

A.3.3 Programs

$<program>$ ::= `program` $<name>$
$<declare>$
$<always>$
$<initial>$
$<assign>$
`end`

$<declare>$ ::= `declare` ($<declItem>$;)*

$<declItem>$ ::= `var` $<declSymbols>$: $<type>$
| `type` $<declSymbols>$ `=` $<type>$

$<declSymbols>$ ::= $<name>$ (, $<name>$)*

$<always>$ ::= `always` ($<defItem>$;)*

$<defItem>$ ::= $<name>$: $<type>$ `=` $<expr>$

$<initial>$ ::= `initially` ($<expr>$;)*

$<assign>$ ::= `assign` ($<statement>$)*

```
<statement>     ::=  <statLabel>  <simpleStat>

<statLabel>     ::=  []
                |    [  <name>  ]

<simpleStat>    ::=  <assignment>  (||  <assignment>)*

<assignment>    ::=  <lhs>  :=  <rhs>

<lhs>           ::=  <var>  (,  <lvalue>)*

<lvalue>        ::=  <name>
                |    <lvalue>  .  <name>
                |    <lvalue>  .  <expr>

<rhs>           ::=  <exprList>
                |    <condRhs>  (~ <condRhs>)*

<condRhs>       ::=  <exprList>  if  <expr>
```

A.3.4 Properties

```
<property>      ::=  constant  <expr>
                |    invariant  <expr>
                |    stable  <expr>
                |    transient  <expr>
                |    <expr>  co  <expr>
                |    <expr>  ensures  <expr>
                |    <expr>  unless  <expr>
```

| | *<expr>* `-->` *<expr>*

| | *<expr>* `-->` *<expr>* `by` *<regexp>*

<regexp> ::= `[]`

| | `[` *<name>* `]`

| | `(` *<regexp>* `)`

| | *<regexp>* `+` *<regexp>*

| | *<regexp>* *<regexp>*

| | *<regexp>* `*`

A.3.5 Parse Units

<input> ::= $(<unit>)^*$

<unit> ::=

| | *<scopedUnit>* `;`

| | `in` *<name>* `:` *<scopedUnit>* `;`

| | *<program>* `;`

| | *<declItem>* `;`

<scopedUnit> ::= *<expr>*

| | *<property>* `;`

A.4 Future Extensions to the Input Language

Two important features of the UV input language not implemented in version 2.3.3 are *quantified formulae, statements, and assignments* and *variable ordering directives.*

In order to introduce quantification to the language we need to add a quantifier construct:

$<quant>$ ::= $<quantDummies>$: $<type>$ | $<quantRange>$:

$<quantDummies>$::= $<nameList>$

$<quantRange>$ ::=
| $<expr>$

The range of a quantification starting with such a quantifier construct consists of a type for the dummy variables and an optional boolean expression restricting the values of the dummies to those satisfying the expression.

Quantified statements are introduced by modifying the grammar rule for the statement non-terminal $<statement>$ as follows:

$<statement>$ ::= $<statLabel>$ $<simpleStat>$
| $<quantStat>$

$<quantStat>$ ::= ([] $<quant>$ $(<statement>)^*$)

Similarly, quantified assignments are introduced by modifying the assignment non-terminal $<assignment>$:

$<assignment>$ ::= $<lhs>$:= $<rhs>$
| (|| $<quant>$ $<simpleStat>$)

Finally, quantified expressions are obtained by adding the following alternative to the rule for the expression non-terminal $<expr>$:

$<expr>$ ::= ($<op>$ $<quant>$ $<expr>$)

where $<op>$ stands for any commutative and associative binary operator with a unit element.

As far as directives for influencing the variable ordering for the OBDD representation of program variables are concerned it might be argued that such information should be separated from the actual program text, as it constitutes information not so much about the program but about how to make it manageable for a specific symbolic representation. Therefore, we plan to include commands for managing variable orderings as separate entities that can be loaded, modified, browsed, and saved. However, until this will be implemented, a simple mechanism that gives the user some additional control over the variable ordering seems desirable.

Such a mechanism is the one that has been implemented in an earlier revision (1.19) of the UV system, in which the user can cause certain variables to be interleaved by tagging them with the `%interleaved` directive. This mechanism is included by extending the grammar rule for the declare section as follows:

```
<declare>      ::= declare (<ditem> ;)*

<declItem>     ::= var <declSymbols> : <type> <interl>
                 | type <declSymbols> = <type>
<declSymbols>  ::= <name> (, <name>)*

<interl>       ::=
                 | %interleaved
                 | %interleaved <name>
```

The directive `%interleaved` after a list of variables causes all variables in the list to have interleaved BDD indices. The directive `%interleaved` followed by a variable name after a list of variables causes all the variables in the list to have BDD indices that are interleaved not only for the list of variables, but also for the named variable (and all variables the named variable is interleaved with). *<name>* in the last variation must be a previously declared variable name (including the

immediately preceding list of declared variables), there is no restriction on its type (in particular it does not need to match the type of the declared variables).