View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by ScholarlyCommons@Penn

University of Pennsylvania
ScholarlyCommons

Technical Reports (CIS) Department of Computer & Information Science

October 1995

# VERSA: Verification, Execution and Rewrite System for ASCR

Duncan Clarke
*University of Pennsylvania*

Follow this and additional works at: https://repository.upenn.edu/cis_reports

## Recommended Citation
Duncan Clarke, "VERSA: Verification, Execution and Rewrite System for ASCR", . October 1995.

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-95-34.

This paper is posted at ScholarlyCommons. https://repository.upenn.edu/cis_reports/196
For more information, please contact repository@pobox.upenn.edu.

VERSA: Verification, Execution and Rewrite System for ASCR

## Abstract

VERSA is a tool for the automated analysis of resource-bound real-time systems using the Algebra of Communicating Shared Resources (ACSR). This document serves as an introduction to the tool for beginning users, and as a reference for process and command syntax, examples of usage, and tables of operators, built-in functions and algebraic laws. Two detailed examples demonstrate the application of VERSA to cononical examples from the literature.

This version of the VERSA user's guide reflects the 95.09.10 version of the tool.

## Comments

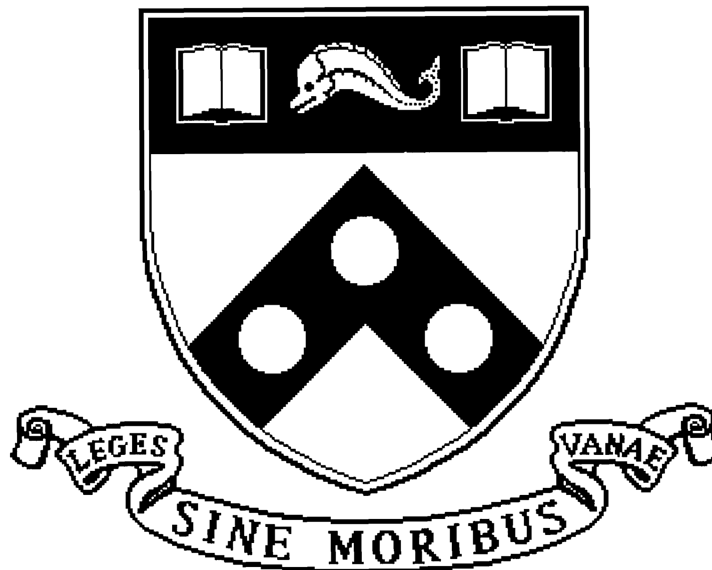
University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-95-34.

This technical report is available at ScholarlyCommons: https://repository.upenn.edu/cis_reports/196

# VERSA: Verification, Execution and Rewrite System for ACSR [1]

MS-CIS-95-34

Duncan Clarke




University of Pennsylvania
School of Engineering and Applied Science
Computer and Information Science Department
Philadelphia, PA 19104-6389

1995

[1]This research was supported in part by ONR N00014-89-J-1131s1, NSF CCR-9415346, and AFOSR F49620-95-1-0508

# VERSA: Verification, Execution and Rewrite System for ACSR *

Duncan Clarke
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104-6389

October 30, 1995

**Abstract**

VERSA is a tool for the automated analysis of resource-bound real-time systems using the Algebra of Communicating Shared Resources (ACSR). This document serves as an introduction to the tool for beginning users, and as a reference for process and command syntax for users of all experience levels. Coverage includes a complete description of process and command syntax, examples of usage, and tables of operators, built-in functions and algebraic laws. Two detailed examples demonstrate the application of VERSA to canonical examples from the literature.

This version of the VERSA user's guide reflects the 95.09.10 version of the tool. The latest version of VERSA is available by anonymous ftp from ftp.cis.upenn.edu in directory pub/rtg, and through the World-Wide Web via the Penn Real-Time Group home page http://www.cis.upenn.edu/~rtg/home.html.

*This research was supported in part by ONR N00014-89-J-1131S1, NSF CCR-9415346, and AFOSR F49620-95-1-0508.

# Contents


**1 Introduction** **3**

**2 General Syntax** **3**
- 2.1 Format . . . 3
- 2.2 Comments . . . 3
- 2.3 Identifiers . . . 3
- 2.4 Reserved Words . . . 3

**3 Basic Data Types** **4**
- 3.1 Integer Constants . . . 4
- 3.2 Event Label Constants . . . 4
- 3.3 Resource Name Constants . . . 4
- 3.4 Process Constants . . . 4

**4 Composite Data Types** **4**
- 4.1 Set Constants . . . 4
- 4.2 Action Constants . . . 4
- 4.3 Event Constants . . . 4
- 4.4 Pair Constants . . . 4

**5 Operators and Expressions** **4**
- 5.1 Expressions . . . 4
- 5.2 Index Definitions . . . 5
- 5.3 Operand Notation . . . 5
- 5.4 Arithmetic (→*integer*) . . . 5
- 5.5 Relational (→*integer*) . . . 6
- 5.6 Boolean (→*integer*) . . . 6
- 5.7 Miscellaneous (→*integer*) . . . 6
- 5.8 Sets (→*set*) . . . 6
- 5.9 Prefix (→*process*) . . . 7
- 5.10 Composition (→*process*) . . . 7
- 5.11 Context (→*process*) . . . 7
- 5.12 Miscellaneous (→*process*) . . . 8
- 5.13 Precedence and Associativity . . . 8

**6 Commands** **9**
- 6.1 Miscellaneous . . . 9
- 6.2 Binding Process Variables . . . 9
- 6.3 Queries . . . 9
- 6.4 Process Equivalence Checking . . . 10
- 6.5 Process Interpretation . . . 10
- 6.6 Interpreter Commands . . . 11

**7 Preprocessor** **11**
- 7.1 Token Replacement . . . 12
- 7.2 Macros . . . 12
- 7.3 File Inclusion . . . 12
- 7.4 Conditional Compilation . . . 12
- 7.5 Pragmas . . . 12

**A X-Windows Interface** **14**

**B Built-In Functions** **14**
- `min(`*ie1,...,ien*`)` . . . 14
- `max(`*ie1,...,ien*`)` . . . 14
- `sqr(`*ie*`)` . . . 14
- `sqrt(`*ie,[round]*`)` . . . 14
- `UniqueDigits(`*n,radix*`)` . . . 14
- `LeadingDigit(`*n,radix*`)` . . . 14
- `TrailingDigit(`*n,radix*`)` . . . 14
- `HasDigit(`*n,d,radix*`)` . . . 14
- `rand(`*ceil*`)` . . . 14

**C Algebraic Laws** **15**
- Choice1···Choice7 . . . 15
- Par1···Par6 . . . 15
- Scope1···Scope6 . . . 16
- Res1···Res6 . . . 16
- Close1···Close6 . . . 16
- Rec1 . . . 16

**D Examples** **17**
- D.1 Two Bit Buffers . . . 17
- D.2 The Jobshop . . . 18

# 1 Introduction

This paper is intended as an introduction to the VERSA system for beginning users and a reference for advanced users. VERSA is a toolkit for modeling systems using ACSR, the Algebra of Communicating Shared Resources. ACSR is a real-time process algebra that incorporates the interleaved event synchronization model of CCS and a synchronous semantics for time passage steps. Novel aspects of ACSR include explicit notions of resources, time, and priority.

VERSA facilitates the construction and analysis of real-time systems using ACSR with the following features:

- Support for ACSR's full syntax and semantics.
- Syntax and semantic checking of process expressions.
- Support for indexed process names, event labels, and resource names.
- Generalized operators for economically expressing operations on indexed process names, event labels, and resource names.
- Manipulation of ACSR process terms according to a set of laws preserving strong bisimulation.
- Equivalence checking and refutation for pairs of processes.
- Interactive execution of the labeled transition system corresponding to an ACSR process.

The sections that follow present a complete description of VERSA's input syntax. ACSR and VERSA semantics are treated informally. This paper is a quick reference for VERSA. It is not intended as a tutorial or formal treatment of ACSR.

# 2 General Syntax

## 2.1 Format

- Spaces, tabs, newlines and formfeeds are used as separators. Extra such characters are legal and can be used to improve readability.

## 2.2 Comments

- Two types:
  1. Begin with `/*`, end with `*/`.
  2. Begin with `//`, end with newline.
- Legal anywhere a space is legal.

## 2.3 Identifiers

- Identifiers are used as the names of process variables, event labels, resource names, index variables, and functions.
- Legal characters — `a-z A-Z 0-9` underscore (`_`), apostrophe (`'`)
  - First character must be alphabetic.
  - There is no limit on identifier length.
  - Any apostrophes must occur last.
- Examples — `P  P2  P_i  Delay  P'  P2''`

## 2.4 Reserved Words

- Process Components

| | | | |
|---|---|---|---|
| `and` | `idle` | `inf` | `infinite` |
| `infinity` | `infty` | `NIL` | `or` |
| `rec` | `scope` | `t` | `tau` |

  The sequence of characters *NIL* is reserved for every capitalization of its individual letters.

- Generalized Process and Set Operators

| | | |
|---|---|---|
| `Choice` | `Parallel` | `Set` |
| `Union` | `Intersect` | `Complement` |

- Algebraic Law Names

| | | | |
|---|---|---|---|
| `Choice#` | `Par#` | `Scope#` | `Res#` |
| `Close#` | `Rec#` | `Rhide#` | `Relab#` |

  The # symbol denotes any sequence of digits. The first letter may be upper or lower case.

- Commands

| | | |
|---|---|---|
| `bindings` | `bye` | `ctsmp` |
| `debug` | `echo` | `exit` |
| `fold` | `guarded` | `quit` |
| `terse` | `unbind` | `unbindall` |
| `unfold` | `unwind` | `verbose` |
| `whynot` | | |

# 3 Basic Data Types

The basic data types are integer (decimal), event label, resource name, and process. This section presents the syntax for constants.

## 3.1 Integer Constants

- Digits `0-9` or the keyword **`infty`**.
- Keyword **`infty`** is a special symbol representing $\infty$. Only allowed context is the time bound in a `scope()` operator.
- Keyword **`infty`** has aliases **`inf`**, **`infinite`**, and **`infinity`**.
- Examples — `007   12   scope(P,e,infty,Pe,NIL,Pi)`

## 3.2 Event Label Constants

- Identifiers.
- Optionally prefixed with an apostrophe (`'`). Corresponds to bar over events in traditional process algebra notation, as in $\overline{e}$.
- Optionally suffixed by a comma separated list of integer indices enclosed in square brackets (`[` and `]`).
- The keywords **`tau`** and **`t`** represent the distinguished event label $\tau$.
- Examples — `in   'Out   tau   e[1,1]`

## 3.3 Resource Name Constants

- Identifiers.
- Optionally suffixed by a comma separated list of integer indices enclosed in square brackets (`[` and `]`).
- Examples — `cpu   Printer   cell[34]`

## 3.4 Process Constants

- `NIL` is the only process constant.
- Deadlocked process; performs no events or actions.

# 4 Composite Data Types

The built-in composite data types are set, action, event, and pair. This section presents the syntax for constants.

## 4.1 Set Constants

- Unordered comma separated list of homogeneous elements.
- Allowed element types are event labels, resource names, pairs, and resource, priority pairs.
- Examples — `{rd,wrt}   {}   {(r,1),(s,1)}`

## 4.2 Action Constants

- Set of parenthesis enclosed resource name, priority pairs.
- Priority is expressed as an integer value.
- The keyword **`idle`** is an alias for the action `{}`.
- Examples — `idle   {}   {(r1,5),(r2,7)}`

## 4.3 Event Constants

- Parenthesis enclosed event label, priority pair.
- Priority is expressed as an integer value.
- Examples — `(e,1)   (in[34],27)   (tau,0)`

## 4.4 Pair Constants

- Slash (/) separated pair of event labels or resource names.
- Examples — `P/p[34]   R/R'`

# 5 Operators and Expressions

## 5.1 Expressions

- An *expression* consists of one or more operands with an operator.
- Parenthesis may be used freely to improve readability or override default operator precedences.
- Examples —
  - `i+7`
  - `P+NIL`
  - `R[x==(y*z)/w] || S`

## 5.2 Index Definitions

- An index is an identifier that represents an integer variable.
- The range of an index is defined by an index definition.
- Syntax — {*var*, (*max* | *min*, *max*[, *step*[, *cond*]])}

  var — The index being defined.

  min — An integer expression for the initial value. (Default is 1).

  max — An integer expression for the maximum value.

  step — An integer expression for the value by which the index is incremented. (Default is 1).

  cond — Boolean conditional evaluated for each value of the index. If *cond* is false the index value is not used. (Default is 1).

- Example — `{i,1,100,j,i%2 == 0}`
  This index ranges from 1 to 100 by steps of $j$. (Index $i$ is a sub-index of an index $j$, the value of which is used for the increment of $i$.) Only even values of $i$ will be available to the context of this index definition.
- An index definition can define multiple indices. A comma-separated list of individual index definitions is used. Index values are initialized and computed from *left-to-right.*
- Example — `{i,1,10},{j,1,i}`
  Index $i$ ranges from one to ten, and for each value of $i$, index $j$ ranges from one up to the value of $i$.

## 5.3 Operand Notation

- Some operators require specific kinds of operands. The following notation is used to indicate differences.

  $e$ Any expression or constant.

  $v$ Any expression that refers to a variable to which a value can be assigned.

- A prefix indicates expression type. For example, *ie* is any integer expression. The complete list of type prefixes follows:

  $i$ — integer

  $p$ — process

  $l$ — event label

  $e$ — event

  $r$ — resource name

  $a$ — action

  $d$ — index definition

- Pairs of a given label type are indicated by a $p$ suffix, for example *lep* for an event label pair or *rep* for a resource name pair.
- Sets of a given base type are indicated by an $s$ suffix, for example *les* for an event label set.
- If several operands appear in an expression, then they may be distinguished by appending numbers, for example *pe1* + *pe2*.

## 5.4 Arithmetic (→*integer*)

Addition: `+`

- Usage — *ie1* + *ie2*
- Example — `i+1`

Subtraction: `-`

- Usage — *ie1* - *ie2*
- Example — `j-1`

Negation: `-`

- Usage — - *ie*
- Example — `-j`

Multiplication: `*`

- Usage — *ie1* * *ie2*
- Example — `i*2`

Division: `/`

- Usage — *ie1* / *ie2*
- Example — `j/2`

Remainder: `%`

- Usage — *ie1* % *ie2*
- Example — `k%2`

Exponentiation: `**`

- Usage — *ie1* ** *ie2*
- Example — `m**3`

## 5.5 Relational (→*integer*)

The logical value "false" is represented by integer 0, and "true" by any non-zero value. The values of relational and boolean expressions are 0 for false and 1 for true.

Equal: `==`

- Usage — *ie1* `==` *ie2*
- Example — `{i,0,10,1,(j%2)==0}`

Not Equal: `!=`

- Usage — *ie1* `!=` *ie2*
- Example — `{i,0,10,1,i!=j}`

Less Than: `<`

- Usage — *ie1* `<` *ie2*
- Example — `{i,0,10,1,i<j}`

Less Than or Equal: `<=`

- Usage — *ie1* `<=` *ie2*
- Example — `{i,0,10,1,i<=j-1}`

Greater Than: `>`

- Usage — *ie1* `>` *ie2*
- Example — `{i,0,10,1,2*i>j}`

Greater Than or Equal: `>=`

- Usage — *ie1* `>=` *ie2*
- Example — `{i,0,10,1,2*i>=j-1}`

## 5.6 Boolean (→*integer*)

Negation: `!`

- Usage — `!` *ie*
- Example — `{i,0,10,1,!(i<j+k)}`

Disjunction: `or`

- Usage — *ie1* `or` *ie2*
- Example — `{i,0,10,1,(i<3) or (i>6)}`

Conjunction: `and`

- Usage — *ie1* `and` *ie2*
- Example — `{i,1,9,1,(i%2) and (i<j)}`

## 5.7 Miscellaneous (→*integer*)

Function Call: *FuncName*`()`

- Usage — *FuncName*`(`*ie1*`,...,`*ien*`)`
  The built-in function indicated by *FuncName* (see Appendix B) is applied to the argument list.
- Example — `Set[e[sqrt(i)] i,1,100]`

## 5.8 Sets (→*set*)

Set Generator: `Set[]`

- Usage — `Set[`*le1*`,...,`*len nd*`]`
  Generates a set of event labels.
- Example — Set[e[i] {i,1,5}]

- Usage — `Set[`*re1*`,...,`*ren nd*`]`
  Generates a set of resource names.
- Example —
  `Set[r[i,j],s[i] {i,1,5},{j,1,5}]`

- Usage — `Set[`*lep1*`,...,`*lepn nd*`]`
  Generates a set of event label pairs.
- Example —
  `Set[in[i+2]/e[i] {i,1,5}]`

- Usage — `Set[`*rep1*`,...,`*repn nd*`]`
  Generates a set of resource name pairs.
- Example —
  `Set[cpu[i*2]/r[i] {i,1,5}]`

Union: `Union[]`

- Usage — `Union[`*es1*,...,*esn*`]`
  Forms the union of two or more sets containing elements of the same base type.
- Example —
  `Union[{r},Set[s[i] {i,1,5}]]`

Intersection: `Intersect[]`

- Usage — `Intersect[`*es1*,...,*esn*`]`
  Forms the intersection of two or more sets containing elements of the same base type.
- Example —
  `Intersect[{r[1],r[2],r[3]},{r[2],s}]`

Complement: `Complement[]`

- Usage — `Complement[`*es1*,*es2*`]`
  Forms the complement of set *es1* with respect to universe *es2*. The elements of *es1* and *es2* must be of the same base type.
- Example —
  `Complement[{r[3]},Set[r[i] {i,1,5}]]`

## 5.9 Prefix (→*process*)

Event Prefix: `.`

- Usage — *ee*`.`*pe*
  A process that synchronizes on *ee* and continues as the process *pe*.
- Example — `(e,1).P`

Action Prefix: `:`

- Usage — *ae*`:`*pe*
  A process that executes the time-consuming action *ae* and continues as the process *pe*.
- Example — `{(r,1),(s,2)}:Q`

## 5.10 Composition (→*process*)

Choice: `+`

- Usage — *pe1* `+` *pe2*
  A process that chooses to continue as *pe1* or *pe2* depending on one or more of the following: (1) event and resource offerings of the environment; (2) priority arbitration; and (3) nondeterministic choice among alternatives.
- Example — `(e,1).P + Q`

Generalized Choice: `Choice[]`

- Usage — `Choice[`*pe de*`]`
  The choice process that results from composing the process expressions *pe* that result for index values defined by *de*. Similar to the following ACSR notation: $\sum_{i=min}^{max} P_i$.
- Example — `Choice[P[i] {i,1,10}]`

Parallel Composition: `||` (or `|`)

- Usage — *pe1* `||` *pe2*
  The process that result from executing *pe1* and *pe2* simultaneously. Events are interleaved or synchronize to produce $\tau$ events. Time consuming actions must execute concurrently.
- Example — `((e,1).P1+{}:P2)||Q`

Generalized Parallel Composition: `Parallel[]`

- Usage — `Parallel[`*pe de*`]`
  The choice process that results from composing the process expressions *pe* that result for index values defined by *de*. Similar to the following ACSR notation: $\prod_{i=min}^{max} P_i$.
- Example —
  `Parallel[(e[i],1).Q {i,1,10}]`

## 5.11 Context (→*process*)

Temporal Scope: `Scope()`

- Usage — `scope(pe1,le,ie,pe2,pe3,pe4)`
  Process *pe1* is executed for up to *ie* time units. If *pe1* executes an event labeled with *le* before *ie* time units elapse then the scope is terminated and the process continues as *pe2*. If *pe1* is executed for exactly *ie* time units without the scope being terminated the scope is terminated and the process continues as *pe3*. The scope can be terminated at any time before *ie* time units elapse by executing an event or action offered by *pe4*. In that case, the process continues as *pe4*.
- Example —

```
scope(rec X.{}:X,
      dummy,10,
      NIL,TimesUp,NIL)
```

Resource Closure: `[]`

- Usage — `[pe]`*res*
  The process formed by augmenting every action *a* of *pe* with *(r,0)* for every $r \in res$ such that *a* has no resource name, priority pair with resource *r*.
- Example — `[{(r,1),(t,3)}:P]{r,s,t}`
  The first action of this process would be `{(r,1),(t,3),(s,0)}`. Subsequent actions of `P` would be augmented in the same way.

Event Restriction: `\`

- Usage — *pe* `\` *les*
  The process formed by prohibiting *pe* from executing events labeled with event labels from *les*.

- Example —
  `((e,1).P || ('e,1).Q)\{e,f,g}`
  The process can only perform the action $(\tau,2)$ and continue as $(P||Q)\backslash\{e,f,g\}$ because the offered events are restricted.

Resource Hiding: `\\`

- Usage — *pe* `\\` *rns*
  The process formed by eliminating all resource, priority pairs labeled with resource names from *rns* from the actions of *pe*.
- Example — `({(r,1),(s,1)}:P)\\{s}`
  The first action of this process would be `{(r,1)}`. Subsequent actions of `P` would be modified in the same way.

Total Resource Hiding: `\\{*}`

- Usage — *pe* `\\ {*}`
  The process formed by eliminating all resource, priority pairs from the actions of *pe*.
- Example — `({(r,1),(s,1)}:P)\\{*}`
  The first action of this process would be {}. Subsequent actions of `P` would be modified in the same way.

Relabeling: `%`

- Usage — *pe* `%` `[`*leps*`,`*reps*`]`
  The events of *pe* are relabeled according to the pairs of *leps* and the resources names appearing in actions of *pe* are relabeled according to the pairs of *reps*.
- Example —
  `((e,1).{(r,1)}:P)%[{init/e},{CPU/r}]`
  The first event of this process would be `(init,1)`. The action following this event would be `{(CPU,1)}`. Subsequent events and actions of `P` would be modified in the same way.

### 5.12 Miscellaneous (→*process*)

Recursion: `rec`

- Usage — `rec` *pv.pe*
  Standard recursion on process variable *pv*.
- Example —
  `rec X.((again,1).X + (stop,1).NIL)`

Law Application: *LawName*`()`

- Usage — *LawName*`(`*pe*`)`
  The law indicated by *LawName* (see Appendix C) is applied to process *pe*.
- Example — `Choice1(((e,1).P)+NIL)`
  **Law *Choice1* eliminates `NIL` from choice, so the resulting process is `(e,1).P`.**

Folding: `fold()`

- Usage — `fold(` *pe* `,` *pv* `)`
  Every occurrence of the process bound to *pv* that occurs as a subprocess of *pe* is replaced by *pv*.
- Example — `fold((e,1).P'+Q,P)`
  If the process `(e,1).P'` is bound to `P` then the result of this folding operation is the process `P+Q`.

Unfolding: `unfold()`

- Usage — `unfold(` *pe* `,` *pv1*`,...,`*pvn* `)`
  For each process variable *pv* in *pv1,...,pvn*, every occurrence of *pv* in *pe* is replaced by the process expression bound to *pv*.
- Example — `unfold(P+Q,P)`
  If the process `(e,1).P'` is bound to `P` then the result of this unfolding operation is the process `(e,1).P'+Q`.

### 5.13 Precedence and Associativity

- Precedence — In the chart below, the operators within a group have equal precedence. Higher precedence operator groups are higher in the chart.
- Associativity — In the absence of explicit parentheses, associativity rules are used to determine how to group operators and operands (left-to-right, or right-to-left), when the operators are in the same group.
- Examples —
  - `a*b/c` is equivalent to `(a*b)/c` because of left-to-right associativity.
  - `(e,2).{}:P` and `(e,2).({}:P)` are equivalent because of right-to-left associativity.

| | | |
|---|---|---|
| ** | Exponentiation | LEFT-TO-RIGHT |
| * | Multiply | LEFT-TO-RIGHT |
| / | Divide | |
| % | Remainder | |
| \ | Event restriction | LEFT-TO-RIGHT |
| . | Event prefix | *RIGHT-TO-LEFT* |
| : | Action prefix | |
| \|\| | Composition | LEFT-TO-RIGHT |
| + | Addition, choice | LEFT-TO-RIGHT |
| - | Subtract | |
| < | Less than | LEFT-TO-RIGHT |
| > | Greater than | |
| <= | Less than or equal | |
| >= | Greater than or equal | |
| == | Equal | LEFT-TO-RIGHT |
| != | Not equal | |
| and | Logical and | LEFT-TO-RIGHT |
| or | Logical or | LEFT-TO-RIGHT |

# 6 Commands

## 6.1 Miscellaneous

Termination:

- Usage: `quit`, `exit`, or `bye`.

Display Mode:

- Usage: `terse`
  Set terse output mode. Eliminates all non-essential messages.
- Usage: `verbose`
  Set verbose output mode. Enable all user-oriented messages.
- Usage: `debug`
  Set debug output mode. Enables all user-oriented messages and generates copious amounts of data useful for debugging the VERSA system.
- Usage: `echo`
  Toggles echoing of input lines. If *echo* mode is on all input is copied to standard output. If *echo* mode is off input lines read from `#include`*'ed* files will not be displayed.

## 6.2 Binding Process Variables

Simple:

- Usage — *pv* = *pe* ;
  The process *pe* is bound to the process variable *pv*. If *pv* is already bound the old binding is saved and can be restored with an `unbind` command.
- Example — `P = (e,1).P';`

Generative:

- Usage — *pv* = *pe de* ;
  Multiple bindings may be registered, depending on how many index values are generated by *de*. Process variable *pv* must be indexed by exactly the index variables defined by *de*.
- Example —
  ```
  Q[i] = ('psYes,1).NIL
    {i,1,100,1,sqrt(i)*sqrt(i)==i};
  ```
  This process generator binds only those `Q[i]` for which `i` is a perfect square.

Unbinding:

- Usage — `unbind` *pv* ;
  The current binding of process variable *pv* is removed. If *pv* was already bound at the time of its most recent binding, the old binding is restored. Binding and unbinding implement push and pop operations on a LIFO stack of bindings with its head bound to *pv*.
- The keyword *unwind* is an alias for *unbind*.

- Usage — `unbindall;`
  Remove all bindings.

## 6.3 Queries

General Information:

- Usage — ?
  Displays a general help message summarizing commands and syntax.

Bindings:

- Usage — `bindings ?`
  Displays all current process bindings.

Identifier Types:

- Usage — *ident* ?
  Displays the type (if any) of the identifier. If the identifier is a bound process variable its binding is displayed.

Event/Action Comparisons:

- Comparison operators (*cop*) are `==`, `!=`, `<`, `<=`, `>`, and `>=`.
- Usage — (*ee1* | *ae1*) *cop* (*ee2* | *ae2*) ?
  The preemption relation is applied to the pair of operands to determine whether one preempts the other. That result is conditioned by the comparison operator and an appropriate message is output.
- Example — `(e,3) > (e,1)`
  A message is output indicating that the query is true. Event `(e,3)` preempts event `(e,1)`.

Guardedness:

- Usage — **guarded(** *pe* **,** *pv* **)** ?
  Tests whether all occurrences of process variable *pv* are guarded by prefix operators in process *pe* and outputs an appropriate message.

- Usage — **guarded(** *pe* **)** ?
  Tests whether every process variable that occurs in *pe* is guarded by a prefix operator and outputs an appropriate message.

## 6.4 Process Equivalence Checking

Equivalence:

- Usage — *pv1* == *pv2* ?
  The processes bound to *pv1* and *pv2* are compared to determine whether they are equivalent according to any of the following notions of equivalence:
  1. *Identity:* The abstract syntax tree representations of the two processes are compared node by node to determine whether the two processes have identical syntax. Time to complete is linear in the size of the abstract syntax trees.
  2. *Unique Fixpoint Induction:* The abstract syntax tree representations of the two processes are compared to determine whether the two processes have identical structure with the exception of process variable naming and referencing. Time to complete is proportional to the size of the abstract syntax trees.
  3. *Prioritized Strong Equivalence:* Labeled Transition System (LTS) models are constructed for each of the processes and a state minimization algorithm is applied to the two state machines to determine whether or not they are bisimilar. Time to complete is exponential in the size of the abstract syntax trees for finite state processes. For infinite state processes the test does not terminate.
  4. *Prioritized Weak Equivalence:* LTS models are constructed for each of the processes. All $\tau$-labeled event edges are removed according to an algorithm that mimics the algorithm for computing the $\epsilon$-closure of a finite state automaton. A state minimization algorithm is applied to the two $\tau$-free LTS's to determine whether or not they are bisimilar. Time to complete is exponential in the size of the abstract syntax trees for finite state processes. For infinite state processes the test does not terminate.

Refutation:

- Usage — **whynot** ?
  If the most recent equivalence test returned a result of false for the prioritized strong equivalence test this command will output a shortest path to the first state in each LTS where the processes diverge.

- Usage — **whynot^** ?
  If the most recent equivalence test returned a result of false for the prioritized weak equivalence test this command will output a shortest path to the first state in each $\tau$-free LTS where the processes diverge.

## 6.5 Process Interpretation

Executing the LTS:

- Usage — *pv* !
  An LTS is constructed for the process bound to *pv* and interpreter mode is entered. The commands accepted in interpreter mode are described in Section 6.6.

Executing the $\tau$-free LTS:

- Usage — *pv*^ !
  An LTS is constructed for the process bound to *pv*. All $\tau$-labeled event edges are removed according to an algorithm that mimics the algorithm for computing the $\epsilon$-closure of a finite state automaton. Interpreter mode is entered for the $\tau$-free LTS. The commands accepted in interpreter mode are described in Section 6.6.

## 6.6 Interpreter Commands

Interpreter commands are accepted in a special mode that is activated by the commands listed in Section 6.5. Interpreter mode allows the user to interactively step through the LTS (or $\tau$-free LTS) corresponding to a process. The commands available in interpreter mode are listed below. The default value for optional numeric parameters is one.

General:

- `?` or `help`—Display a general help message summarizing commands and syntax.
- `quit`—Exit interpreter mode.

Edge Traversal:

- `step` [*edge*]—Advance along edge number *edge* to a new node.
- `back` [*steps*]—Backtrack by *steps* edge traversals.
- `cont` [*edge*]—Starting with *edge*, advance along edges until a choice arises or the trace length limit is met.
- `rand`—Advance along edges making choices according to a uniformly distributed random variable whenever a node with multiple output edges is encountered. Continue until the trace length limit is met.
- `seed` *seedval*—Seed the random number generator with *seedval*.
- `limit` *steps*—Set the trace length limit to *steps*.
- `limit`—Set the trace length limit to $\infty$.

Interpreter State Management:

- `trace`—Display the current trace.
- `trace^`—Display the current trace without listing $\tau$ labeled events.
- `clear`—Clear the current trace without changing the interpreter's current node.
- `save`—Save a pointer to the current node and a copy of the current trace to a LIFO stack.
- `restore`—Reset the execution state to the state and trace `save`*ed* most recently.

Queries:

- `show`—Display the outgoing edges of the current node.
- `show` *edge*—Display the outgoing edges of the node reachable via edge number *edge*.
- `show limit`—Display the current value of the trace length limit.
- `show stack`—Display stack of `save`*ed* nodes and traces.
- `show stats`—Display statistics for the reachable state space of the LTS including node count, edge count, counts of deadlocked states, zeno states, states capable of stopping the clock, and CPU time to compute the LTS.
- `show time`—Display a count of time consuming steps contained in the current trace.
- `show deadlock`[s]—Display the process term for each deadlocked node and a shortest path from the start node to each deadlocked node.

# 7 Preprocessor

A `#` as the first character on a line designates a preprocessor control line. Control lines are terminated by a newline. Use a backslash just before the newline to continue a control line.

## 7.1 Token Replacement

`#define` *identifier string*

- Example — `#define DELAY 10`

  Substitutes `10` for every occurrence of `DELAY` as a token.

`#undef` *identifier*

- Example — `#undef DELAY`

  Cancels previous `#define` for identifier `DELAY`, if any.

## 7.2 Macros

*Note*—To avoid precedence conflicts, enclose a macro parameter in parenthesis everywhere it occurs in a macro definition.

`#define` *identifier1(identifier2,...) string*

- Example — `#define isodd(N) (((N)%2)==1)`

  Substitutes `(((arg)%2)==1)` for `isodd(arg)`, replacing each occurrence of `N` by `arg`.

- Example —

  ```
  #define pause(T,NEXT) \
            scope(rec X.{}:X, \
                  dummy,(T), \
                  NIL,(NEXT),NIL)
  ```

  The backslashes are used to continue the macro definition across multiple lines. This macro creates a process expression that executes the time consuming action $\emptyset$ for `T` time units and continues as `NEXT`.

## 7.3 File Inclusion

`#include` *<filename>*

- Example — `#include <stdlib.acsr>`

  Replaces this line with the contents of the file `stdlib.acsr`. The angle brackets specify that `stdlib.acsr` should be found in a directory found in the path defined by the `ACSRLIB` environment variable. Does *not* search the current working directory.

`#include` *"filename"*

- Example — `#include "spec.acsr"`

  Replaces this line with the contents of the file `spec.acsr`. When `"spec.acsr"` is used instead of `<spec.acsr>`, `spec.acsr` is sought in the current working directory.

## 7.4 Conditional Compilation

Conditional compilation control lines are used to compile different code depending on externally defined conditions. Conditional compilation blocks can be nested freely.

`#ifdef` *identifier*

- Example — `#ifdef MODE`

  True if `MODE` is currently defined by `#define`.

`#ifndef` *identifier*

- Example — `#ifndef MODE`

  True if `MODE` is *not* currently defined by `#define`.

`#else`

...

`#endif`

- If preceding `#ifdef` or `#ifndef` test is —
  - True — Lines between `#else` and `#endif` are ignored.
  - False — Lines between the test and a `#else` or, lacking a `#else`, the `#endif`, are ignored.
- `#endif` terminates the conditional compilation.
- Example —

  ```
  #ifndef STDLIB_ACSR
  #include <stdlib.acsr>
  #endif
  ```

## 7.5 Pragmas

Pragmas implement debugging directives that produce printed output describing the internal state of the lexical analyzer or parser.

`#pragma` *identifier string*

- The following identifiers are recognized:

  `symtab_dump` — Display the contents of the symbol table.

  `mactab_dump` — Display the contents of the macro table.

  `text` — Parse *string* as though it were regular input.

  `sanity_test` — Test integrity of processes.

  `msg` — Copy *string* to standard error output.

## References


[1] D. Clarke, I. Lee, and H. Xie. VERSA: A tool for the specification and analysis of resource-bound real-time systems. *Journal of Computer and Software Engineering*, 3(2), April 1995.

[2] R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench: A Semantics-Based Tool for the Verification of Concurrent Systems. *TOPLAS*, 15:36–72, 1993.

[3] Formal Systems (Europe) Ltd., 3 Alfred Street—Oxford OX1 4eH—UK. *Failures Divergence Refinement: User Manual and Tutorial*, April 1993.

[4] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[5] I. Lee, P. Brémond-Grégoire, and R. Gerber. A Process Algebraic Approach to the Specification and Analysis of Resource-Bound Real-Time Systems. *Proceedings of the IEEE*, 82(1):158–171, January 1994.

[6] Insup Lee, Duncan Clarke, and Hong-Liang Xie. The algebra of communicating shared resources and its toolkit. In Sang H. Song, editor, *Advances in Real-Time Systems*, chapter 12, pages 275–298. Prentice Hall, 1995.

[7] H. Lin. Pam: A process algebra manipulator. In *Proc. Third Workshop on Computer Aided Verification, LNCS 575.* Springer Verlag, July 1991.

[8] H. Lin. Pam user manual. Technical Report 9/91, School of Cognitive and Computing Sciences, University of Sussex, 1991.

[9] R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

# A X-Windows Interface

An X-Windows interface has been implemented on top of the VERSA implementation described in the main body of this paper. It supports only the unindexed portion of the ACSR process syntax.

The most valuable features of the X-Windows interface are (1) a point-and-click rewrite system that significantly improves the usability of the law application operators described in the main body of this paper; and (2) a point-and-click interface to the interpreter that displays a small portion of the LTS and allows the user to select the edges that are traversed with the mouse pointer.

The X-Windows interface includes a help function that provides complete information on its use.

# B Built-In Functions

This section describes the VERSA system's built-in functions for operating on integer values.

- Minimum: `min(` *ie1,...,ien* `)`
  Computes the minimum of its list of two or more arguments.

- Maximum: `max(` *ie1,...,ien* `)`
  Computes the maximum of its list of two or more arguments.

- Square: `sqr(` *ie* `)`
  Computes the square of *ie*.

- Square Root: `sqrt(` *ie* [ *round* ] `)`
  Computes the integer square root of *ie*, rounding according to *round*. If *round* is zero $\lfloor\sqrt{ie}\rfloor$ is returned. If *round* is one $\lceil\sqrt{ie}\rceil$ is returned. If *round* is not specified $\sqrt{ie}$ is truncated.

- Count Unique Digits: `UniqueDigits(`*ie1,ie2*`)`
  Computes the count of unique digits in *ie1* interpreted as a radix *ie2* integer.

- Leading Digit: `LeadingDigit(`*ie1,ie2*`)`
  Computes the leading digit of *ie1* interpreted as a radix *ie2* integer.

- Trailing Digits: `TrailingDigits(`*ie1,ie2*`)`
  Interprets *ie1* as a radix *ie2* integer and returns all but the first digit.

- Test for Digit: `HasDigit(`*ie1,ie2,ie3*`)`
  Interprets *ie1* as a radix *ie3* integer and checks whether it contains digit *ie2*. Returns one if true, zero if false.

- Uniform Random Number: `rand(`*ie1*`)`
  Returns a uniformly distributed random integer from the interval $[0, ie1)$.

# C Algebraic Laws

| | |
|---|---|
| Choice1 | $P + \text{NIL} = P$ |
| Choice2 | $P + P = P$ |
| Choice3 | $P + Q = Q + P$ |
| Choice4 | $(P + Q) + R = P + (Q + R)$ |
| Choice5 | $A_1 : P_1 + A_2 : P_2 = A_2 : P_2$ if $A_1 \prec A_2$ |
| Choice6 | $(a_1, n_1).P_1 + (a_2, n_2).P_2 = (a_2, n_2).P_2$ if $(a_1, n_1) \prec (a_2, n_2)$ |
| Choice7 | $A : P + (\tau, n).Q = (\tau, n).Q$ if $n > 0$ |
| Par1 | $\text{NIL}\Vert\text{NIL} = \text{NIL}$ |
| Par2 | $(A : P)\Vert\text{NIL} = \text{NIL}$ |
| Par3 | $(a, n).P\Vert\text{NIL} = (a, n).(P\Vert\text{NIL})$ |
| Par4 | $P\Vert Q = Q\Vert P$ |
| Par5 | $(P\Vert Q)\Vert R = P\Vert(Q\Vert R)$ |
| Par6 | see below |

Par6

$$
\Big(\sum_{i\in I} A_i : P_i \;+\; \sum_{i\in J}(a_j, n_j).Q_j\Big) \;\Vert\; \Big(\sum_{k\in K} B_k : R_k \;+\; \sum_{l\in L}(b_l, m_l).S_l\Big)
$$

$$
= \left[\begin{array}{l}
\displaystyle\sum_{\substack{i\in I, j\in J,\\ \rho(A_i)\cap\rho(B_j)=\emptyset}} (A_i \cup B_j) : (P_i\Vert Q_j) \\
\displaystyle + \sum_{j\in J} (a_j, n_j).(P_j\Vert(\sum_{k\in K} B_k : R_k \;+\; \sum_{l\in L}(b_l, m_l).S_l)) \\
\displaystyle + \sum_{l\in L} (b_l, m_l).((\sum_{i\in I} A_i : P_i \;+\; \sum_{j\in J}(a_j, n_j).Q_j))\Vert S_l \\
\displaystyle + \sum_{\substack{j\in J, l\in L,\\ a_j=b_l}} (\tau, n_j + m_l).(Q_j\Vert S_l)
\end{array}\right.
$$

Table 1: ACSR Laws (Part 1 of 2)

| | |
|---|---|
| Scope1 | $A : P \triangle^b_t (Q, R, S) = A : (P \triangle^b_{t-1} (Q, R, S)) + S$ if $t > 0$ |
| Scope2 | $(a, n).P \triangle^b_t (Q, R, S) = (a, n).(P \triangle^b_t (Q, R, S)) + S$ if $t > 0 \wedge a \neq b$ |
| Scope3 | $(a, n).P \triangle^b_t (Q, R, S) = (\tau, n).Q + S$ if $t > 0 \wedge a = b$ |
| Scope4 | $P \triangle^b_0 (Q, R, S) = R$ |
| Scope5 | $(P_1 + P_2) \triangle^b_t (Q, R, S) = P_1 \triangle^b_t (Q, R, S) \; + \; P_2 \triangle^b_t (Q, R, S)$ |
| Scope6 | $\mathrm{NIL} \triangle^b_t (Q, R, S) = S$ |
| Res1 | $\mathrm{NIL} \backslash F = \mathrm{NIL}$ |
| Res2 | $(P + Q) \backslash F = (P \backslash F) + (Q \backslash F)$ |
| Res3 | $(A : P) \backslash F = A : (P \backslash F)$ |
| Res4 | $((a, n).P) \backslash F = (a, n).(P \backslash F)$ if $a \notin F \wedge \overline{a} \notin F$ |
| | $((a, n).P) \backslash F = NIL$ if $a \in F \vee \overline{a} \in F$ |
| Res5 | $(P \backslash F_1) \backslash F_2 = P \backslash (F_1 \cup F_2)$ |
| Res6 | $([P]_I) \backslash F = [P \backslash F]_I$ |
| Close1 | $[\mathrm{NIL}]_I = \mathrm{NIL}$ |
| Close2 | $[P + Q]_I = [P]_I + [Q]_I$ |
| Close3 | $[A_1 : P]_I = (A_1 \cup A_2) : [P]_I$ where $A_2 = \{(r, 0) \vert r \in I - \rho(A_1)\}$ |
| Close4 | $[(a, n).P]_I = (a, n).[P]_I$ |
| Close5 | $[[P]_I]_J = [P]_{I \cup J}$ |
| Close6 | $[P \backslash F]_I = ([P]_I) \backslash F$ |
| Rec1 | $recX.P = P[recX.P/X]$ |

Table 2: ACSR Laws (Part 2 of 2)

# D Examples

## D.1 Two Bit Buffers

```
TBB  = (in,1).TBB1;
TBB1 = (in,1).TBB2 + (out,1).TBB;
TBB2 = (out,1).TBB1;

SYS  = (OBBL||OBBR)\{sync};
OBBL = (in,1).(sync,2).OBBL;
OBBR = ('sync,2).(out,1).OBBR;
```

Processes TBB and SYS provide two alternative formulations of a two bit buffer. They are adapted from a CCS example found in [9]. TBB is a purely sequential formulation of a two bit buffer that receives one or two inputs by synchronizing on in and generates output by synchronizing on out. If zero bits are currently held only in is possible. If one bit is held either in or out is possible. If two bits are held the buffer is full so only out is possible. SYS is an alternative formulation that uses concurrency instead of explicit sequential structure to realize the buffer.

VERSA can be used to demonstrate the equivalence of these two formulations as follows:

```
:-) #include "2bb.acsr"
:-) TBB == SYS?
    ufi failed--following pair could not be matched:
        <(in,1).TBB1,(OBBL || OBBR)\{sync}>
              --following pair was matched:
        <TBB,SYS>
    false (by prioritized strong equivalence)
    true (by prioritized weak equivalence)
:-) whynot?
prefix:  --(in,1)-->
unmatched TBB:
 --(in,1)-->
 --(out,1)-->
unmatched SYS:
 --(tau,(2+2))@sync-->
```

The "#include" command reads the process descriptions from a file named 2bb.acsr. The "TBB == SYS?" command initiates equivalence checking of the TBB and SYS processes. The first result that is printed is the outcome of the unique fixpoint induction test. The test fails because TBB and SYS disagree on their respective first operators. The next result is the outcome of the prioritized strong equivalence test. Since that test failed, the next result that is printed is the outcome of the prioritized weak equivalence test, which succeeded. Processes TBB and SYS are equivalent by prioritized weak equivalence (the notion of bisimulation that disregards internal $\tau$ events).

The "whynot?" command demonstrates the equivalence refutation feature. It lists edges that can be traversed to arrive at states that are not bisimilar according to the definition of prioritized strong equivalence. The output shows that after an in event TBB and SYS differ because TBB is ready for another in or out immediately, but SYS must first synchronize internally on its sync event.

## D.2 The Jobshop

```
#define JOBRANGE 1,3

#define EASY  1
#define HARD  2
#define OTHER 3

#define DONE(JOB) 0

#define IFELSE(PRED,P1,P2) ((tau,2*((PRED)!=0)+1).(P1) + (tau,2).(P2))

Sem  = {}:Sem + (get,1).rec Sem'.((put,1).Sem  + {(Semaphore,1)}:Sem');

HamSem = Sem%[{geth/get,puth/put},{Hammer/Semaphore}];
MalSem = Sem%[{getm/get,putm/put},{Mallet/Semaphore}];

Jobber = Choice[(in[job],1).Start[job] {job,JOBRANGE}] + {}:Jobber;
Start[job] = IFELSE(job==EASY,{}:Finish[job],
                              IFELSE(job==HARD,Usehammer[job],
                                               Usetool[job])) {job,JOBRANGE};
Usetool[job] = Usehammer[job] + Usemallet[job] +
               (tau,1).{}:Usetool[job]                      {job,JOBRANGE};
Usehammer[job] = ('geth,1).{}:{}:('puth,1).Finish[job] +
                 {}:Usehammer[job]                          {job,JOBRANGE};
Usemallet[job] = ('getm,1).{}:{}:('putm,1).Finish[job] +
                 {}:Usemallet[job]                          {job,JOBRANGE};
Finish[job] = ('out[DONE(job)],1).Jobber                    {job,JOBRANGE};

Jobshop = [Jobshop']{Hammer,Mallet};
Jobshop' = (Jobber || Jobber || HamSem || MalSem)\{geth,puth,getm,putm};
```

Process **Jobshop** is an ACSR implementation of the job shop example found in [9]. The **#define** of **JOBRANGE** defines the range of values that represent jobs. The next three **#define**s break **JOBRANGE** into three distinct types of jobs, **EASY**, **HARD**, and **OTHER**. The **DONE** macro translates a job into a completed job.

The **IFELSE** macro uses ACSR's priority relation to define an if/then/else construct. If the value of the boolean predicate **PRED** is non-zero (i.e. true) then process **P1** is prefixed with a priority $2*1+1=3$ internal event. Otherwise process **P1** is prefixed with priority $2*0+1=1$ internal event. The "else" process **P2** is always prefixed with a priority 2 internal event. According to the semantics of the choice operator if **PRED** is true $(\tau,3).P_1$ will be executed, otherwise $(\tau,2).P_2$ will be executed.

Process **Sem** is a general purpose binary semaphore with idling. Initially **Sem** can idle for one time unit or receive a **get** request to allocate the semaphore. Once the semaphore is allocated it remains allocated and any idling steps include the **Semaphore** resource until a **put** request is received. Process **HamSem** uses renaming of events and resources to create a **Hammer** resources whose access is controlled by **geth** and **puth** events. Process **MalSem** creates a similar **Mallet** resource.

Process **Jobber** describes a single worker in the job shop. A job is received by synchronizing on an `in[job]` event, or the jobber idles for one time unit. If a job of type **job** was received, the process continues as `Start[job]`. Note how generalized choice (i.e. `Choice[···]`) and indexing are used to simulate value passing with indexed events. **Start** evaluates the job type and (1) completes "easy" jobs in one time unit without the use of any tools; (2) proceeds as **Usehammer** if the job is "hard;" or (3) proceeds as **Usetool** if the job falls somewhere in between.

Process `Usetool` attempts to use either the hammer or the mallet to complete the job, depending on which tool (if any) is available. If no tool is available, `Usetool` idles for one time unit and tries again. Processes `Usehammer` and `Usemallet` allocate and use their respective tools, or idle one time unit if the tool is not immediately available. Process `Finish` marks the job as complete and outputs the job with an `out` event.

Process `Jobshop` creates a job shop with two jobbers by composing `Jobber` processes in parallel with `Hammer` and `Mallet` controlling processes. The `get` and `put` operations for the two resources are restricted to insure that the resources can only be allocated locally, and no external resources are allocated. The `Jobshop` is closed in the `Hammer` and `Mallet` resources to insure that `Hammer` and `Mallet` are used exclusively by the two `Jobber` processes.