ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΠΑΝΕΠΙΣΤΗΜΙΟ
ΚΥΠΡΟΥ

<u>Παραδοτέο 5</u>: Οδηγός Χρήσης Εργαλείου για ανάλυση πιθανοτικών συστημάτων

Ερευνητικό Πρόγραμμα ΕΝΔΙΚΤΗΣ

VERSA: Verification, Execution and Rewrite System for QoSPA

Abstract

VERSA is a tool that assists in the algebraic analysis of real-time systems. It is based on QoSPA, a real timed process algebra designed to express resource-bound real-time distributed and networked systems. VERSA supports the analysis of real-time processes through long-run average characteristics, interactive execution, and Quality of Service characteristics testing. 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, as well as the state-based analysis features.

Table of Contents

1. Introduction

Reliability in real-time systems can be improved through the use of formal methods for the specification and analysis phases. Formal methods treat system components as mathematical objects and provide mathematical models to describe and predict the observable properties and behaviours of these objects. There are several advantages to using formal methods for the specification and analysis of real-time systems. Some of them are:

- The early discovery of ambiguities, inconsistencies and incompleteness in informal requirements;
- The automatic or machine-assisted analysis of the correctness of specifications with respect to requirements; and
- The evaluation of design alternatives without expensive prototyping.

Despite all the virtues of using formal methods, their manual application tends to be time consuming and error prone. To alleviate this strain, we have built VERSA (Verification, Execution, and Rewrite System for QoSPA), an integrated toolkit whose goal is to simplify the specification and analysis of resource-bound real-time systems.

Based upon the algebra QoSPA, VERSA is a tool that supports an algebraic approach to systems analysis and design. Systems are described as algebraic expressions, and they can be analyzed by

- Construction of a state machine, and subsequent exploration and analysis of the state space of that machine to verify safety properties such as freedom from deadlocks and equivalence of alternative process formulations; or
- Interactive execution of the process specification to explore specific system properties and sample the execution traces of the system; or
- Analyze the long run average behaviour of specific system properties; or
- Analyze the Quality of Service characteristics of the system.

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

- Support for QoSPA'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.
- Interactive execution of the labelled transition system corresponding to a QoSPA process.

The sections that follow present a complete description of VERSA's input syntax. This paper is a manual for VERSA. It is not intended as a tutorial or formal treatment of QoSPA. Section 2 presents an overview of the architecture of VERSA. Sections 3-8 present the syntax of the tool. Finally in Section 9 we present our conclusions and some future work.

2. The VERSA environment

The VERSA environment has been designed to allow the integrated of use state-space exploration analysis and performance evaluation. The design has been implemented using object-oriented techniques and the C++ language to facilitate maintenance, enhancement, and portability. The sections that follow describe the VERSA system.

Section 2.1 provides an overview of the system architecture.

2.1 The VERSA architecture

Figure 1 shows the overall architecture of the VERSA system. The architecture is divided into three major areas: front-end, intermediate representation, and analysis.

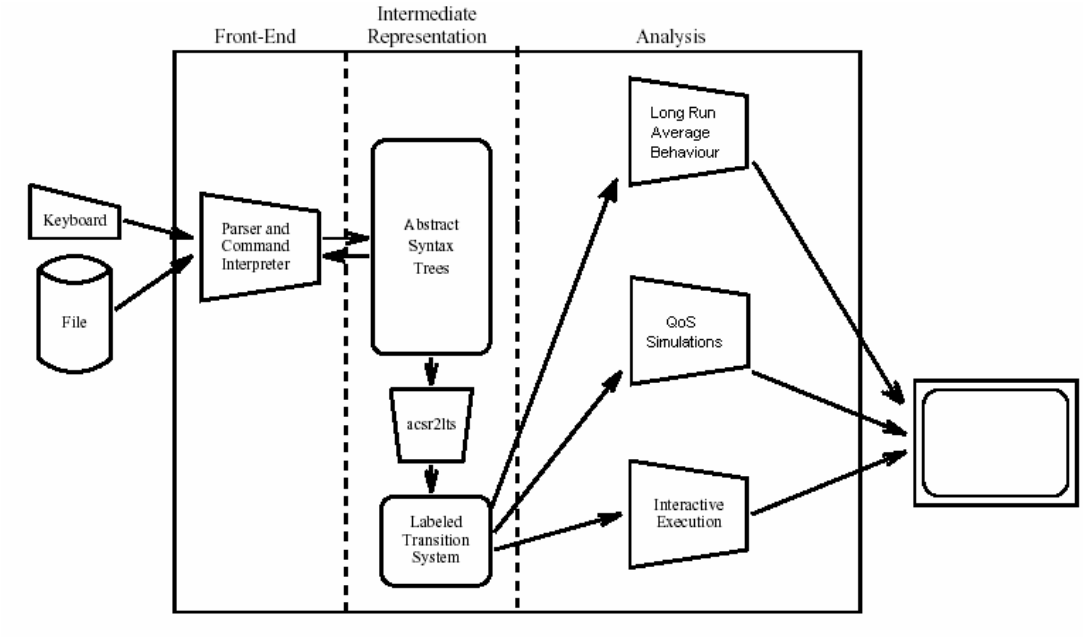

Figure 1: VERSA System Structure

The front-end is responsible for management of input sources, parsing of input commands and process definitions, and presentation of a consistent interface to the user. The current user interface is keyboard and file-based, allowing commands and process descriptions to be entered directly from the keyboard, input from a file, or a combination of keyboard and file input. The lexical analysis and parsing of the input is handled by a Lex/Yacc based parser.

At present there are three major functions implemented:

- Quality of service simulations.
- Long run average behaviour
- Interactive execution.

All of the above functions operate on a labelled transition system (LTS). The LTS for one or more processes is produced by an algorithm that expands the process to produce a labelled transition system representing all possible executions. The LTS construction algorithm also prunes edges made unreachable by the semantics of the

prioritized transition system, in most cases reducing the size of the resulting state machine.

Quality of service (QoS) simulations uses the classic method of simulations in order to find out the quality of service that a given system supports. Given a network topology and a simulation time, we can simulate the network behaviour and report some QoS characteristics, like throughput and drops.

The long run average behaviour system is used in order to find the average behaviour of the characteristics of a system if that system is running for so long time that it tends to be infinite. The characteristics that are under investigation are the ones that refer to robustness and reliability, like mean delay time and network fairness. This method gives an alternative to the simulations method in order to analyze the attributes-quality of the systems.

The interactive execution feature allows user-directed execution of process specifications. The user may interactively step through the LTS one action at a time, produce traces from random executions of the LTS, save process configurations to a stack for later analysis while an alternate path is explored, and analyze the size and deadlock characteristics of the LTS resulting from their process.

3. Syntax of VERSA input

3.1 Format

Spaces, tabs, new lines and form feeds are used as separators. Extra such characters are legal and can be used to improve readability.

3.2 Comments

Two types:

1. Begin with /*, end with */.
2. Begin with //, end with new line.

A space is legal anywhere.

3.3 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"

3.4 Reserved Words

- Process Components
  and idle inf infinite infinity infty NIL 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

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

4. Basic Data Types

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

4.1 Integer Constants

- Digits 0-9 or the keyword infty.
- Keyword infty is a special symbol representing ∞. 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)

4.2 Event Label Constants

- Identifiers.
- Optionally prefixed with an apostrophe ('). Corresponds to bar over events in traditional process algebra notation, as in 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 t.
- Examples: in 'Out tau e[1,1]

4.3 Resource Name Constants

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

4.4 Process Constants

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

5. Composite Data Types

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

5.1 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)}

5.2 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)}

5.3 Event Constants

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

6. Operators and Expressions

6.1 Expressions

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

6.2 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 j 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.

6.3 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.

6.4 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 | f(r,1),(s,2)g:Q

6.5 Composition (→process)

Choice: +

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

Parallel Composition: || (or |)

- Usage | pe1 || pe2 The process that result from executing pe1 and pe2 simultaneously. Events are interleaved or synchronize to produce τ events. Time consuming actions must execute concurrently.
- Example: ((e,1).P1+fg:P2)||Q

6.6 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 labelled 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.fg:X,dummy,10,NIL,TimesUp,NIL)

**Event Restriction: **

- Usage | pe \ les The process formed by prohibiting pe from executing events labelled with event labels from les.
- Example: ((e,1).P || ('e,1).Q)\{e,f,g}
  The process can only perform the action (τ ,2) and continue as (PjjQ)nfe; f; gg because the offered events are restricted.

6.7 Miscellaneous (→process)

Recursion: rec

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

6.8 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

* / %	Multiply Divide Remainder	LEFT-TO-RIGHT
\	Event restriction	LEFT-TO-RIGHT
. :	Event prefix Action prefix	RIGHT-TO-LEFT
||	Composition	LEFT-TO-RIGHT
+ -	Addition, choice Subtract	LEFT-TO-RIGHT

< > <= >=	Less than Greater than Less than or equal Greater than or equal	LEFT-TO-RIGHT
== !=	Equal Not equal	LEFT-TO-RIGHT
and	Logical and	LEFT-TO-RIGHT
or	Logical or	LEFT-TO-RIGHT

7. Commands

7.1 Miscellaneous

Termination:

- Usage: quit, exit, or bye.
- Display Mode:
  - Echo: Toggles echoing of input lines. If echo mode is on all input is copied to standard output. If echo mode is on all input lines read from #included files will not be displayed.

7.2 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 6.3.

Executing the τ-free LTS:

- Usage: pv* ! An LTS is constructed for the process bound to pv. All τ-labelled event edges are removed according to an algorithm that mimics the algorithm for computing the τ-closure of a _nite state automaton. Interpreter mode is entered for the τ-free LTS. The commands accepted in interpreter mode are described in Section 6.3.

7.3 Interpreter Commands

Interpreter commands are accepted in a special mode that is activated by the commands listed in Section 6.35. Interpreter mode allows the user to interactively step through the LTS (or _ -free LTS) corresponding to a process. The commands available in interpreter mode are listed below. The default value for optional numeric parameters is 1.

General:

- ? or help: Display a general help message summarizing commands and syntax.
- quit or exit: interpreter mode.

Edge Traversal:

- step [edge]: Advance along edge number edge to a new node.
- back [steps]: Backtrack by steps edge traversals.

- how: Display the outgoing edges of the current node.
- show edge: Display the outgoing edges of the node reachable via edge number edge.
- show deadlock[s]|Display the process term for each deadlocked node and a shortest path from the start node to each deadlocked node.

Long Run Average Behaviour:

- compute reset *reset_labels* outcome *outcome_labels gain*: finds the average behaviour of the characteristics of a system if that system is running for so long time that it tends to be infinite
    - i.e. compute reset *overflow lose* outcome *sent 1*

QoS Characteristics:

- throughput simulation_time sample_time, unit_time, packet_size, queue_type, buffer_size, label, filename: finds the throughput of a system run
    - throughput *5 0.1 0.004 1000 drop_tail 200 1b throughput_3.*

8. 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.

8.1 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.

8.2 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.

9. Summary and Future Directions

We have presented VERSA, a tool for interactive specification and analysis of resource-bound real-time systems.

Enhancements of the system include the use of BDDs (Binary Decision Diagrams) or other similar data structure for compaction of a system's state space. Future objectives also include the extension of the tool for a stochastic extension of QoSPA. Other desirable features would be the addition of a database that would allow the VERSA environment to be saved between analysis sessions, implementation of a graphical user interface based on a standard windowing package, and implementation of a model checking facility.