DIMACS Technical Report 97-57
September 1997

Special Year on Logic and Algorithms Tutorial Notes: Complexity Issues in Automata-Theoretic Verification: The COSPAN Approach to Deal with These Issues (Tutorial Lectures by Robert Kurshan)

by

Eric Allender[1]
(Editor, SYLA Tutorial Notes)
Dept. of Computer Science
Rutgers University
New Brunswick, New Jersey 08903

[1]Permanent Member

DIMACS is a partnership of Rutgers University, Princeton University, AT&T Labs, Bellcore, and Bell Labs.

DIMACS is an NSF Science and Technology Center, funded under contract STC–91–19999; and also receives support from the New Jersey Commission on Science and Technology.

ABSTRACT

The 1995-1996 DIMACS Special Year on Logic and Algorithms began with three week-long tutorial sessions on the topics of the special year:

- Finite Model Theory
- Proof Complexity
- Computer-Aided Verification

These notes on the lectures given by Robert Kurshan were compiled by Sandeep K. Shukla.

Additional material from the tutorials may be found from the www page

`http://dimacs.rutgers.edu/Workshops/SYLA-Tutorials/program.html`

A Tutorial Based on the Lectures of Dr. Robert Kurshan at the DIMACS Summer School on Automated Verification in August 1995

Complexity Issues in Automata theoretic Verification: The COSPAN Approach to Deal with these Issues

compiled by
Sandeep K. Shukla [1]

Abstract

This is a tutorial introduction to the COSPAN verification tool with emphasis on the approaches taken in COSPAN to fight the computational complexity lower bounds that tend to limit the applicability of automated verification to large designs.

The material is based on Dr. Kurshan's lectures in the DIMACS summer school on Automated Verification. We acknowledge the contributions of Oleg Sokolsky and Kathy Fisler in the preparation of this tutorial.

For a more complete exposition to the theory one has to read "Computer-Aided Verification of Coordinating Processes : The Automata Theoretic Approach" By Robert P. Kurshan (Princeton University Press, 1994) and the bibliographic references therein.

1 Introduction

1.1 Preamble

In this set of lecture notes, rather than scribing a day to day account of the lectures, we attempt to present a tutorial introduction to the material taught in the lectures. We discuss the complexity theoretic issues that arise in the automata theoretic approach to linear time based verification methodology. In our methodology, computation of a system is viewed as a set of linear sequence of events. Hence, we refer to our methodology as *linear time* based because we have a linear view of the time as opposed to branching time models. The COSPAN tool has been developed specifically to cope with the complexity theoretic lowerbounds that tend to limit the applicability of the automated verification to large designs. Various *refinement* and *abstraction* techniques as well as heuristics for *decomposition* and algorithms for *localizations* have been essential parts of the COSPAN verification tool. As a result, to appreciate the problems related to the computational complexity in automated verification and methods to circumvent them, it will be a great exercise to understand the methodologies and background of this specific tool.

[1]Department of Computer Science, University at Albany, Albany, NY. This notes were written during a summer stay at the Bell Labs.

In this tutorial we assume that the reader is familiar with the basics of ω-automata, linear time temporal logics and the idea of model checking.

We first discuss the ω-automata theoretic approach to verification which is founded in a linear time view of computation and model checking. A hierarchical approach based in *refinement* can be applied to construct verified implementations of specifications. However, adding more details to the implemenation gradually leads to the ever daunting problem of very large state space and the PSPACE-hard lowerbounds. As a result, methodologies which complement the refinement based methods, namely compositionality (or decomposition) and localization have to be developed. This tutorial illustrates such methodological issues with the example of the verification tool [2] which has been developed over a decade with these issues in mind. Moreover, it been successfully used in verifying sufficiently large systems and is going to be a part of a commercial product.

1.2 About ω-automata Based Verification

ω-automata is a well established basis for the specification of concurrent systems. In fact, most of the linear time temporal logic (LTL) verification methodology are based on ω-automata. In order to verify that a given system satisfies a given property or "task", one can specify both the system and the task as ω-automata. Now, the ω-regular language of the system automaton is the set of infinite behaviours of the real system being modeled and the ω-regular language of the "task" automaton are the set of infinite behaviours allowed under the correctness criterion. Hence, a language containment check may be used to verify that the system model is in conformance with the "task". This is well known theory for over a decade and is the basis of most of the *automata theoretic verification* methodologies.

The COSPAN *Cordination Specification Analyser* is a software tool which is also based on this automata theoretic foundation. The automata theoretic method is more general than those based on linear time temporal logic based verification. It is well known that ω-automata are more expressive than LTL and LTL modelchecking problems can be easily encoded in this framework. However, the way these ω-automata are used in modeling the system and the task, in the COSPAN tool is geared towards a top-down verification methodology which not only helps the verification process by closely connecting it to the design structure of the system but also meant to cope with the formidable complexity lowerbounds inherent in most linear time based verification methodologies.

First note that ω-automata are strictly more expressive than propositional linear time temporal logics and hence it makes more sense to use ω-automata as a basis for specification of properties.

The ω-regular languages are defined in terms of finite state automata. However, since in ω-regular case, we talk about infinite strings, the finite state automata defining them cannot have final states in the classical sense. The acceptance of an infinite string is thus defined

[2]COSPAN
is available free of charge for academic purposes from Dr.Robert Kurshan (`k@research.bell-labs.com`)
.

in alternative ways. There are variety of such alternatives and depending on the researchers who defined them, they are called *Buchi* condition , *Muller* condition, *Rabin* condition or *Streett* condition etc. Each such condition carries implications in terms of expressive power and the computational complexity of the associated decision procedures.

In COSPAN, a special type of automata on infinite words are used. We call then $L-$automata and $L-$ processes. They both define ω regular languages. For both of them the acceptance condition is given in terms of a set of *recur* edges and a set of *cycle* sets. The set of *recur* edges is a subset of the set of transitions of the automata. A *cycle* set is a subset of the set of states. And the acceptance condition of an automaton (be it $L-$process or $L-$automaton) is given interms of them.

An infinite sequence is accepted by an $L-$automaton if and only if it has a run of the automaton that either traverses a given recur edge infinitely often or is such that the set of states visited infinitely often is contained in **some** *cycle set.*

However, for an $L-$process the acceptance structure is dual of the above. In this case, a sequence is accepted if and only if it has a run which traverses no recur edge infinitely often, and is such that the set of states visited infinitely often is contained in **no** *cycle set.*

Thus, in the case of $L-$process, the acceptance condition may be thought of as an *exception* condition . $L-$ processes provide a natural mechanism for modelling a system and the "exception" condition may be interpreted as a **fairness** property, excepting **unfair** sequences which, for examples, always remain in a set of states (cycle set) or crosses a recur edge infinitely often where the recur edge might correspond to taking a transition without activating an enabled process.

One should note that in most other computational models used in most other verification methdologies, *fairness* is treated externally as an imposed assumption. However, by making distinction between L-process and L-automata in terms of negative and positive acceptance structure, in our methodology, *fairness* is built into the model.

On the other hand, $L-$automata provide a natural mechanism to model system properties ("tasks") which are to be verified. For example, a *liveness* or *eventuality* property may be defined in terms of sequences which traverse a given set of *recur* edges infinitely often. Suppose we consider checking the property that a process is granted access to a shared resource infinitely often (a liveness property) provided it does not remain in a set of states forever where it doesnot request the resource (fairness property). We then impose the fairness condition on an $L-$process S, which models the system, and express the liveness property interms of an $L-$automaton T. Now the verification of the property under the fairness assumption boils down to the ω-language containment $\mathcal{L}(S) \subseteq \mathcal{L}(T)$.

So the complexity issues that we need to concentrate on is the complexity lower and upper bounds on this problem. However, it is well known that if T is deterministic, then this containment can be decided in time linear in the number of transitions in S and T.

Moreover, it is easy to show that, for every ω-regular language $\mathcal{L}$, one can define a finite set of deterministic $L-$automata $T_1, T_2, ...$, such that

$$\mathcal{L} = \cap_i \mathcal{L}(T_i)$$

Finding this set of T_i's is called the *Task decomposition* which is a major way of fighting the computational hardness problem in our framework. Note that once a suitable decomposition is found, then checking the property boils down to checking that

$$\mathcal{L}(S) \subseteq \mathcal{L}(T_i), \forall i$$

.

An even further step towards coping with the complexity problem is *localization*. The *localization* step replaces each test $\mathcal{L}(S) \subseteq \mathcal{L}(T_i)$ with a simpler test $\mathcal{L}(S_i') \subseteq \mathcal{L}(T_i')$ where S_i' is a reduction of S relative to T_i. This means that some portions of the process S may be completely irrelavant to the task T_i and hence an abstraction of S to S_i' is made. On the other hand, T_i' is an abstraction of T_i consistent with the abstraction of S relative to T_i.

Now, note that *finding a good decomposition is an open problem* in this field. Rightnow, from the knowledge of the process S and the task T, one can guess a decomposition. Then the verification tool may try to verify, if the decomposition is not right then the verification fails. The localization can be done algorithmically because it amounts to identifying the variables that are not relevant with respect to the task and the "freeing" those variables (to be explained later).

Note that the reduction described above is complementary to a refinement based design and verification. In a refinement based verifiable design methodology, one starts with a specification and then designs a process that implements the specification at a very high level of abstraction and as one keeps refining it by adding more and more details to the implementation, one has to keep verifying that the new design is faithful to the previous abstract implementation.

However, eventually an implementation is a collection of processes P_i which may correspond to different components of a system. Now, one has to define a suitable semantics for the collection of $L-$processes and then the actual system P's semantics is given in terms of a compositional semantics of the individual components.

Now notice that the size of the $L-$process corresponding the real system is exponential in the size of the component P_i's. Hence, first composing them and then verifying $\mathcal{L}(P) \subseteq \mathcal{L}(T)$ may be very expensive. However, the knowledge of the compositional structure of P might help in the task decomposition and localization and hence can cut down on the verification task from multiplicative to additive cost measures.

In this set of lectures, first we will consider a brief description of the COSPAN tool and describe the various features of the tool relevant to the above mentioned issues. Then we will go on to more general theory pertaining to reductions, abstractions and refinement that applies to automata theoretic verification in general.

One can read the next section first to get an idea of the COSPAN tool and then carry on with the rest of the sections where the theoretical back ground has been explained. Alternatively, one can come back to Section 2, after reading the other theoretical sections.

2 About the COSPAN verification Tool

2.1 Introduction

COSPAN is a general-purpose, UNIX$^{\mathrm{TM}}$-based software tool for *coordination-specification analysis*. Software and hardware systems can be viewed naturally as sets of communicating (or coordinating) process specifications. COSPAN uses automata-theoretic techniques to symbolically analyze coordinating specifications for user-defined properties, or *tasks*. Each test of task performance constitutes a mathematical proof (or disproof), derived through symbolic analysis (*not* through execution or simulation). Typical applications of COSPAN include software development and hardware development for the implementation of control-intensive structures such as communication protocols, analysis of circuits (with arbitrary feedback) at transistor- or gate-level for race conditions and logical correctness, analysis of array processors for functional correctness, and logical analysis of discrete-event models in economics, medicine and strategic planning.

COSPAN displays its full potential when used to develop systems through a formal top-down development procedure based upon successive refinement. Starting at a very abstract level of definition, a system is repeatedly redefined, adding more detail at each level. Tasks may be checked against a system at any level. Logical analysis in conjunction with software and hardware development permits detection of errors during the course of development, from the earliest developmental stages, onward. The successive refinement procedure has the property that a task whose performance is proved at one developmental level is guaranteed performed at all successive levels. The target software or hardware is *automatically* generated from the final, most detailed level of definition. In the case of software, this means production-quality C-code; in the case of hardware, it means a circuit net description to be used as input to a silicon assembler or other layout tool. In both cases, a variety of optimization steps may be performed.

Central to the tractability of COSPAN's analysis algorithms are formal reduction procedures for coping with the typically enormous state spaces associated with virtually all production-oriented coordination problems. Reduction, used in conjunction with top-down development, tends (in a heuristic sense) to render the computational complexity of analysis of coordinating systems as a subexponentially growing function of the system size. Both reduction and analysis are founded in an automata-theoretic semantic model. This model is based upon a mathematical theory of coordination, which serves to justify the validity of the reduction and analysis algorithms.

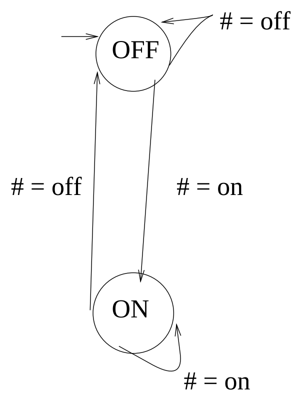
COSPAN programs are written in S/R, an automaton-language which may be used, in each given coordination problem, to build a procedural base supporting declarative structures. In such utilizations, S/R is used to create *ad hoc* abstract data types which define special purpose data-flow languages; because of this feature, S/R is especially suited for developing distributed environments such as communication protocols and highly parallel hardware, in which structure and behavior are naturally understood in terms of data flow. An introduction to S/R is provided in Sect. 2.2

COSPAN supports facilities for documentation, conformance testing and software maintenance, as well as facilities for libraries of abstract data types and reusable pre-tested components, debugging tools for pinpointing the source of task-performance failures, and stochastic analysis, used to test for timing and statistical performance.

COSPAN views both systems and their properties as state machines. An S/R specification is essentially a list of state machine definitions, where the inputs to each machine are comprised of the outputs of several of the other machines. Each machine has a set of possible *states*. At each state, one or more possible outputs or *selections* are defined. Selections differ from conventional state machine outputs in that the latter are assumed to be a deterministic function of the state, whereas selections are nondeterministic outputs, the range of which is a function of the state.

Transition predicates describe how a machine "resolves" a current "global" selection (the vector of all "current" selections of all machines in the specification), by changing state. In a "discrete-event" sense, at all times all machines have some state and some selection "current". Each machine *resolves* the current global selection by moving to a state through a transition whose associated predicate is enabled by the global selection. A system of machines thus evolves through repetition of this procedure whereby each machine "selects" a selection possible from its current state and then each machine "resolves" the current global selection thus formed by moving to a new state through a transition which is enabled by the current global selection. The semantics of this evolution through repeated selection and resolution is independent of the order in which the machines are declared in the S/R specification. Specifically, several machines may change their respective selections at the same time, thus defining the new current global selection. Each change of selections is called a *crank*.

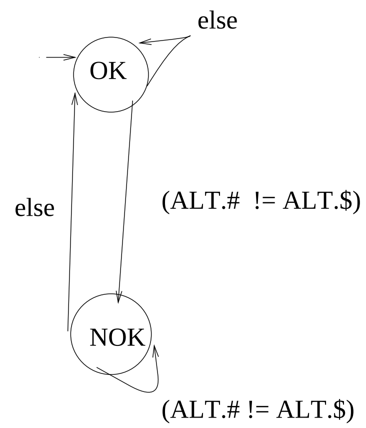
The uniform treatment of systems and properties as state machines gives rise to a verification methodology based on language-containment testing. In illustrating this methodology, we will use diagrammatic representations of state machines, rather than the textual syntax of S/R. Given a state machine M_S representing a system S and a state machine M_P representing a property P, P holds of S iff $\mathcal{L}(M_S) \subseteq \mathcal{L}(M_P)$, where $\mathcal{L}(M)$ denotes the language of state machine M. For example, consider the following state machine, ALT; let $\#$ denote the current selection and let each state be named by the identifier in the corresponding circle.



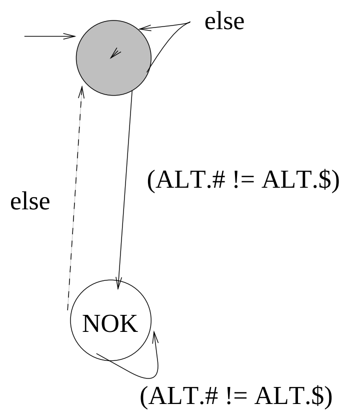
The language of ALT consists of all strings in which the state of ALT follows the selection of ALT in the next crank. Letters in the strings consist of pairs of the form ⟨state, selection⟩. For example, the following string is in $\mathcal{L}$(ALT), where the ω operator is interpreted like the Kleene-star operator for regular languages, except it denotes infinite rather than finite strings:

$$\langle \text{OFF, off}\rangle\langle \text{OFF, on}\rangle\langle \text{ON, off}\rangle\langle \text{OFF, off}\rangle\langle \text{OFF, on}\rangle\langle \text{ON, on}\rangle^{\omega}$$

Suppose we want to test whether the state and selection of ALT correspond infinitely often: i.e., whether every word in L(ALT) infinitely often contains one of the letters ⟨OFF, off⟩ and ⟨ON, on⟩. The following state machine, TASK, expresses the property if all words that eventually cycle forever in state NOK are excluded from its language.



Words are excluded from state machine languages using *acceptance conditions*. For infinite words, acceptance conditions distinguish between *good* and *bad* cycles through the machine. A property specification must indicate the good cycles; any remaining cycles are assumed to be bad. A cycle is indicated by either a set of states or an edge. In the latter case, any cycle in the machine crossing the edge is acceptable. Diagrammatically, shading of states and dashed lines on edges define these cycles. Accordingly, state OK must be shaded and the edge from NOK to OK dashed in the above state machine in order to eliminate words that cycle in state NOK.



In testing L(ALT) for containment in L(TASK), COSPAN searches for words in L(ALT) that remain in bad cycles in TASK. COSPAN locates word

$$(\langle \mathrm{OFF}, \mathrm{on}\rangle\langle \mathrm{ON}, \mathrm{off}\rangle)^\omega$$

which ALT produces by always choosing a selection that forces a state transition. TASK rejects this word because it results in an infinite cycle in state NOK, which is not acceptable.

This small example captures the essence of verification in COSPAN. Systems and properties are both expressed as state machines over infinite words. COSPAN checks whether the language of the system is contained in the language of the property. If not, COSPAN provides a counterexample in the form of a particular word. The next two sections provide more in-depth treatment of system modeling, specification, and proof in COSPAN.

2.2 Modeling Hardware

Consider the S/R representations of ALT and TASK discussed in the previous subsection. Each definition begins with the key word **proc** followed by the machine's name, and ends with the line containing the key word **end**. The characters /* and */ are used to delimit comments (which, as in C, do not nest).

```
proc ALT /*ALTERNATOR*/
    selvar # : (off, on)
    stvar  $ : (OFF, ON)
    init   OFF
    trans

    OFF            {off, on}
          → ON   : # = on
          → OFF  : else;

    ON             {off, on}
          → OFF  : # = off
          → ON   : else;

end /*ALT*/
```

State and selection variables are specified with keywords **stvar** and **selvar** respectively. Here, the selection variable # may take values in the range of string values (off, on), and likewise for the state variable $. While inside ALT these variables may be referenced by their local names # and $, outside ALT they must be referenced by their full path names ALT.# and ALT.$. The symbols '#' and '$' have no special meaning; although they are conventionally used to denote state and selection variables, any valid identifiers may also be used.

The **init** statement defines the set of initial states of the machine (in this case, comprising the single state OFF). In the "transition section" of the declaration (between **trans** and **end**), one sees that two selections are possible from each of the states OFF and ON: $\# = \mathit{off}$ and $\# = \mathit{on}$, as denoted by the *selection predicate* '{off, on}' associated with each of the states. While in the state OFF, if the selection *on* is selected, then

$$\# = \mathit{on}$$

is true, and the transition to state ON is enabled. This is declared in the first *transition declaration*

$$\rightarrow \mathrm{ON} : \# = \mathrm{on}$$

which follows the selection predicate; for all other possibilities (in this case, only $\# = \mathit{off}$) the **else** applies, and the transition to the state OFF is enabled (in which case the resolution in ALT from the "current" state OFF, of the global selection $\# = \mathit{off}$, is a "self-loop" back to OFF). The keyword **else** denotes a Boolean function whose value is the negation of all previous transition predicates in the same transition block.

The following declaration of the TASK machine begins with the key word **monitor** rather than **proc**. A **monitor** is a process which is allowed to use the state variables of other processes in its transition predicates. With that one exception, its syntax and semantics are identical to that of a **proc**. No **proc** or **monitor** may use a variable unless that variable (or a path prefix of that variable's name) is declared in an **import** declaration. The declaration

import ALT

which declares the import of the path prefix ALT, permits TASK to use both variables ALT.# and ALT.$ in its transition predicates. Had TASK been declared as a **proc** rather than a **monitor**, that same **import** declaration would have permitted TASK to use only ALT.#D.

```
monitor TASK
    import   ALT
    stvar    $ : (OK, NOK)
    recur    NOK → OK
    cyset    {OK}
    init     OK
    trans

    true     → NOK : ALT.# ≠ ALT.$
             → OK : else;

end /* TASK */
```

Note that the TASK machine has no **selvar** declaration (and hence, no selection predicate). Semantically, this declares that TASK has a single selection at each state, namely the selection **true**. (Recall that semantically, a selection is a predicate, not the value of a variable.) The **cyset** (for "cycle set") and **recur** for "recurrence edge") statements define the acceptance structure, as discussed in the previous section. Sets of states follow **cyset** and a set of edges follows **recur**. Multiple **cyset** and **recur** definitions are permitted in a single **proc** or **monitor** definition.

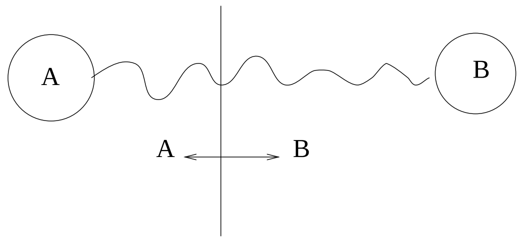
As a more complicated example, consider the following design, which illustrates the modeling of continuous time, delay and transmission across an asynchronous interface. The example specifies a communication protocol comprised of four asynchronous components: a SENDER attempts to send non-null messages (3-bit nonzero bytes, represented by the integers $1, \ldots, 2^3 - 1$) across an asynchronous interface to the outgoing CHANNEL CH[0], and thence on, across another asynchronous interface to the RECEIVER, which then sends an Ack (acknowledgement) across a third asynchronous interface to the incoming CHANNEL CH[1], and on across the fourth asynchronous interface back to the SENDER; this acknowledgement permits the SENDER to send the next message, and so on. Because the SENDER is prevented from sending a second message until it has received the acknowledgement for the first, a flow control protocol is established which should prevent any loss of messages due to buffer overflow in the CHANNEL.

```
type     message : (0..(2^3) − 1)

macro    Ready := 0      /* null message */
         Ack := 1        /* Acknowledgement to SENDER */
         Lost_msg := 8   /* Code for message lost by channel */

proctype CHANNEL( input : integer )
    import          input
    selvar          # : (0..2^3)
    stvar           $ : message
    init            Ready
    trans

    Ready                               {0}
                    → input             : true;

    $ > 0                               {0: $, Lost_msg}
                    → Ready             : (# = $) + (# = Lost_msg)
                    → $                 : else;

    end /* CHANNEL() */

proc SENDER
    import          CH[1]
    selvar          # : message
```

```
    stvar          $ : (SEND, WAIT)
    init           SEND
    trans

    SEND                          {0 : 1..(2^3) − 1}
                   → WAIT         : # > 0
                   → $            : else;

    WAIT                          {0}
                   → SEND         : (CH[1]: Ack)
                   → $            : else;

    end /* SENDER */

    proc CH[i<2] : CHANNEL( SENDER.# ? i = 0 | RECEIVER.# )

proc RECEIVER
    import         CH[0]
    selvar         # : (0,1)
    stvar          $ : message
    init           Ready
    trans

    Ready                         {0}
                   → CH[0].#      : (CH[0]: Lost_msg)
                   → $            : else;

    $ > 0                         {0: Ack}
                   → Ready        : # = Ack
                   → $            : else;

    end /*RECEIVER */
```

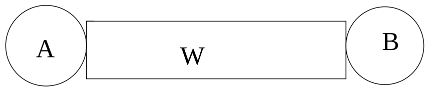
Let us first examine the modeling of time, asynchrony and the syntax of the specification in this example. It is generally helpful to draw diagrams to represent the dataflow in a specification. The flow control protocol of the example may be pictured as

The abutment of the boxes representing the channels to the boxes representing the SENDER and RECEIVER respectively is purposeful, in order to emphasize that signals cross these boundaries in zero time. Unlike other models of coordination in which signals, upon being sent, temporarily "disappear" into the ether only to "reappear" upon being received, in the S/R model, the entire state of the relevant universe is represented, and signals can never "disappear and then reappear". There are two ways to represent the communication between two machines A and B across a wire. One way is to (arbitrarily) apportion a segment of the wire to each, as in



From the vertical line, left, is considered to be A while from the vertical line, right, is considered to be B. In this case, signals cross from A to B in zero time. The exact point of partition is immaterial.

On the other hand, if it is necessary to model a delay in the propagation of a signal from A to B, then the wire must be considered as a third machine. In this case, the diagram would appear as



Here, signals cross from A to the wire W in zero time, and from W to B in zero time. The propagation delay of the signal from A to B is modeled as a delay *in* W. This is the representation in this example, which we now examine directly.

CHANNEL(), declared as a **proctype**, demonstrates machine types in COSPAN. CHANNEL defines a parameterized machine with one integer parameter, *input*, which is imported. When instantiated, this machine moves from its state Ready, under the condition **true**, to the state whose value is that of the instantiated parameter. From the state Ready, one selection is possible, namely 0, as designed by the selection predicate $\{0\}$. Note that Ready is actually a macro definition of the state 0. The selection from any other state is defined in the state transition block beginning with the state predicate "$\$ > 0$". The selection predicate in that block, $\{0 : \$, \text{Lost\_msg}\}$, defines the selection from any state $\$ > 0$ to be initially 0, upon entering \$; thereafter, the selection may either remain 0 or else change to either \$ or Lost_msg. After changing to \$ or Lost_msg, the selection may vascillate between those two, but may never change back to 0 (until, perhaps, the next state transition).

The SENDER sends a *message*, let us say the integer '5', to the RECEIVER across the outgoing CHANNEL CH[0] by selecting '5' from its state SEND. Upon selecting '5', the predicate $\# > 0$ becomes true, and the SENDER state becomes WAIT (from which state its selection is '0'). When the SENDER selection is '5' and the outgoing CHANNEL CH[0] is in state Ready, the next state of CH[0] is the value of the **proctype** parameter *input*, instantiated in the declaration of **proc** CH[$i < 2$] by the expression

SENDER.# ? i=0 | RECEIVER.#

which is equal to SENDER.# if $i = 0$ and otherwise is equal to RECEIVER.# (a syntax inspired by a similar construction in C). Since the CHANNEL array index i satisfies $i = 0$ in the case of CH[0], the value of *input* is SENDER.#, and the new state of CH[0] becomes '5'. While this might have been a "pen-full" to describe, it all happens instantaneously in the model. Upon entering its state '5', CH[0] first selects the selection to the left of the colon in the *selection predicate*

$$\{0 : \$, \text{Lost\_msg}\}$$

namely, the selection #≡ CH[0].#= 0. This selection is meant to be conceived as a "pause" selection at the state '5'. That is, as long as the selection of CH[0] is '0', CH[0] may remain in the state '5' (*i.e.*, with $ ≡ CH[0].$ = 5),selecting '0'. After some nondeterministically determined nonzero period of time, CH[0] changes its selection to either of the other two values possible at state '5', namely # = $ (= 5) or # = Lost_msg ≡ 8 (Lost_msg is defined to be '8' in the **macro** statement). The period of time that # = 0 is true (which, in the model, may be infinite) models the propagation delay of the message '5' through the CHANNEL. Upon changing its selection to '5' (say), the predicate

$$\# = \$$$

becomes true, and CH[0] moves back to state '0' ($=0). However, CH[0].# = 5 being true caused the RECEIVER to move from its state Ready (≡ 0) to its state '5' (= CH[0].#). If, on the other hand, CH[0] changed its selection from '0' to 'Lost_msg', then while CH[0] would nonetheless revert back to state '0', the RECEIVER predicate ~(CH[0]: Lost_msg) would not be enabled and so the RECEIVER would stay in its state 'Ready', thus modeling the loss of the message by the CHANNEL.

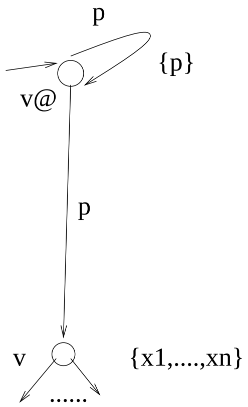
The general syntax of a selection predicate has two possible forms. The first,

$$\{x_1, x_2, \ldots, x_n\}$$

means that any of the selections $x_1, \ldots, x_n$ may be selected from the associated state, and as long as the machine remains in that state, the selections may repeatedly (and nondeterministically) change to any of those n values. The second general form is

$$\{p : x_1, \ldots, x_n\}$$

which means that the machine, upon entering the associated state, call it v, first selects p for some nonzero amount of time. Thereafter the selection may change (nondeterministically in time) to any of $x_1, \ldots, x_n$ and the first case applies (the selection p is no longer possible, unless it is one of the x_i's). The semantics of the second predicate form is shown below:



This predicate form semantically defines two states: v and the "pause" state $v@$ (or, generally, ⟨state name⟩ and ⟨state name⟩@). At the "pause" state $v@$ there is exactly one selection possible, namely the "pause" selection p. Since both the self-loop state transition $v@ \to v@$ and the state transition $v@ \to v$ are enabled by the selection p, either transition may be taken, nondeterministically. However, once the latter is chosen, there may be no (direct) return to $v@$ and the selection of p. From v, the possible selections are $x_1, \ldots, x_n$. While generally the "pause" state $v@$ is suppressed syntactically, occasionally it is necessary to reference $v@$ explicitly, as in

init	$v@$
cyset	$\{v@\}$

and in a *state transition block* such as

```
GO              {p : x, y, z}
     → STOP    :# = x
     → GO@     :# = y
     → $       :else;
```

Here, the state value '\$' is the value of the state variable, that is, the "current" state, the transition to which is enabled if either $\# = p$ (in which case \$ = GO@ or $\# =$ z (in which case \$ = GO). If this discussion of v and $v@$ is confusing, it can be deferred as it is a somewhat technical semantic detail for ensuring the proper treatment of time and is only peripheral to the basic treatment of coordination.

It is important to note the treatment of the asynchronous interface in the example: the machines "synchronize properly" across the four asynchronous interfaces. For example, the outgoing CHANNEL CH[0] initially waits in its state 'Ready' for input from the SENDER, and thus this input cannot be lost as it crosses the interface. On the other hand, if the initial state of CH[0] were to have been (say) 2@ (*i.e.*, selecting the "pause" selection '0' from the state nominally designated as "2"), then the SENDER could select '5', with no effect upon CH[0]; the message '5' would be lost, because of "synchronization" failure, although *not* explicitly lost by the CHANNEL (recall that an explicit CHANNEL loss is modeled by the

CHANNEL selection 'Lost_msg'). Thus, one requirement for the proper performance of this protocol is local synchronization across each asynchronous interface prior to message-passing.

A few additional syntactic details regarding the example have not yet been covered. The reader has surely figured out the usage of the **type** declaration, which permits 'message' to be used as a **type**, in place of the range to its right. If A is a **proc** with selection variable # and # has the string 'Lost_msg' in its range, then the two expressions

$$\text{(A: Lost\_msg)}$$
$$\text{A.\#} = \text{Lost\_msg}$$

are synonymous, both representing the predicate "A selects Lost_msg". The former has the advantage of admitting of an implicit OR construction:

$$\text{(A: Lost\_msg, Pass\_msg)}$$

being synonymous with

$$(\text{A.\#} = \text{Lost\_msg}) + (\text{A.\#} = \text{Pass\_msg})$$

Next, note the *state predicate* in the second *state transition block* of the process TASK. That predicate is

$$\$ \neq \text{Lost\_msg}$$

but the semantic meaning is

$$(\$ \neq 0)^*(\$ \neq \text{Lost\_msg})$$

which could just as well be written in its place. The shorter version is sufficient because state predicates are assumed to be mutually exclusive, and are checked in the order presented in the S/R specification. (This is a bit of the procedural part of the S/R language, which is used for implementational efficiency, with no sacrifice of expressive power.)

Finally, note the use of **else** in this same state transition block. As already stated, the keyword **else** denotes a Boolean function whose value is the negation of all previous transition predicates in the same transition block. (This does not include the transition predicate in which the **else** in question appears.)

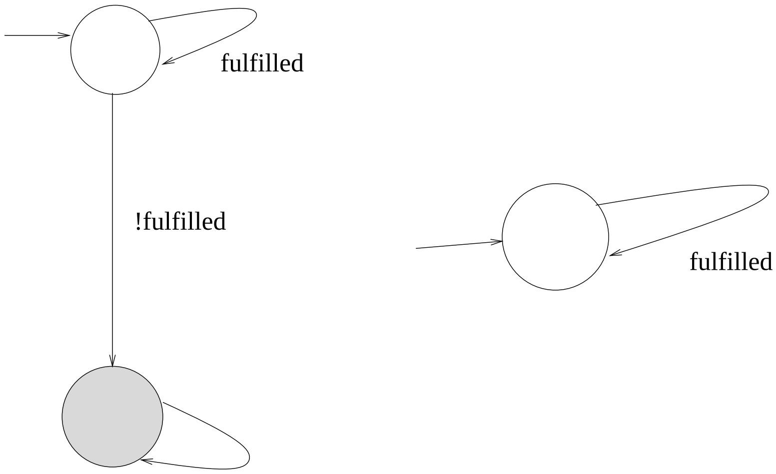
2.3 Specification and Proof

COSPAN uses two types of state machines: processes and automata. Processes are used to specify systems and are viewed as language *generators*. Automata are used to state properties and are viewed as language *acceptors*. Both processes and automata are expressed using the

proc syntax in S/R, as demonstrated in the previous examples.

The difference between processes and automata lies in their acceptance structures. Cycle sets and recur edges are used as a means of defining the acceptance structures for properties. When used to define automata, cysets and recur edges define *accepting* behavior: a word is accepted by an automaton iff it crosses the **recur** edges infinitely often or if it eventually remains in some set of states specified in a **cyset** definition. When used to define processes, cysets and recur edges define *rejecting* behavior: a word is produced by a process iff it does not cross any recur edge infinitely often and if it never cycles forever in any set of states specified in a **cyset** definition.

As an example, consider the definition of a predicate "Always" as each of a process (left) and an automaton (right), where Always(*fulfilled*) holds if there is no reachable state in which *fulfilled* is false.



These two state machines define the same language if the left diagram is interpreted as a process and the right diagram is interpreted as an automaton. COSPAN treats all words as infinite words; as a result, it reports an error if it ever reaches a state from which there is no next-state transition. Therefore, the diagram on the right will accept no word in which *fulfilled* is ever false, because there is no next-state available in that case.

Syntactically, there is no way to indicate whether a state machine should be interpreted as a process or as an automaton. COSPAN interprets machine definitions based on the option flags specified by the user. The user, therefore, writes a specification with particular interpretations in mind. The advantage to this uniform specification style are clear from the operation of the containment test.

Let us return to the ALT and TASK state machines from previous section. ALT and TASK were intended to be interpreted as a process and an automaton, respectively. Recall that the language containment test checks whether $\mathcal{L}(\mathrm{ALT}) \subseteq \mathcal{L}(\mathrm{TASK})$. Mathematically, this is equivalent to testing whether

$$\mathcal{L}(\mathrm{ALT}) \cap \overline{\mathcal{L}(\mathrm{TASK})} = \emptyset$$

where $\overline{\mathcal{L}(M)}$ denotes the complement of the language of M. Operationally, COSPAN tests containment by performing the emptiness test. Therefore, it needs an efficient way to complement TASK. In the state machine models used by COSPAN, interpreting an automaton as a process yields the complement of the automaton (and likewise for processes). Complementation in this framework therefore comes "for free".

Given a file *alt.sr* containing the definitions of both ALT and TASK, the desired containment test is therefore performed if COSPAN interprets both definitions as processes (which complements TASK), takes their intersection, and tests that intersection for emptiness. This sequence of steps is the default behavior of COSPAN, resulting from running

cospan alt.sr

In addition to the default behavior, COSPAN has a flag for treating all state machines as automata; flags also exist that take in two files, where one file contains automata and the other, processes.

2.4 Handling the Complexity Theoretic Lowerbounds

As with any algorithmic verification tool, COSPAN fails to terminate on designs with sufficiently large state spaces. COSPAN supports several techniques for approaching verification in the light of this problem, known as state explosion: abstractions, localizations, and decompositions. Abstractions hide irrelevant design details. Localizations attempt to isolate the portion of the state space that is relevant to a particular property. Decompositions split a property into several smaller properties sufficient to establish the original property. Abstractions are the most common top-down technique, and have been studied in several contexts. Reductions have been studied mainly via on-the-fly approaches to model-checking. Decompositions have received little treatment in the literature. Tools tend to support either abstractions or reductions, but the two are rarely supported in tandem.

COSPAN supports abstraction by providing a homomorphism checker. The user provides two processes and a mapping from the values of one process to the values of the other. COSPAN tests whether the mapping defines a homomorphism between the two processes. If a homomorphism exists, all properties verified of the abstract model are guaranteed to hold of the detailed model.

Localization is COSPAN's version of state-space reduction. Using the variable dependency graph, COSPAN attempts to isolate a sufficient portion of the state space for checking a given property. If the check succeeds in the reduced model, it is guaranteed to succeed in the full model. If it fails, COSPAN attempts to simulate the returned error trace in the full model. If the error exists in the full model, the property does not hold. If the error cannot be reproduced in the full model, COSPAN uses the variable dependency graph to expand the reduced model, and repeats the check. Localization is purely a heuristic technique. The user may seed the localization algorithm with sets of signals to include and exclude from the initial localized model; a precisely-defined seed will often succeed on properties for which

COSPAN fails to find a suitable localization.

Decompositions of properties are user-defined. A decomposition is given in two parts: the original property and the set of automata representing the subproperties. COSPAN tests whether the language of the original automaton is contained in the language of the intersection over the set of subproperty automata. If this test succeeds, the original property is guaranteed to hold if each of the subproperties holds. Decompositions are useful in conjunction with localization, as they divide a larger, "global" property into several, hopefully more localized, properties.

3 The Theoretical Foundations: Boolean Algebra, Languages, and Selection-Resolution

In this section. we discuss the basic setup for the theory of $L-$processes and $L-$automata. A basic knowledge of Boolean Algebra, duality, homomorphisms and finite automata theory and assumed.

3.1 Boolean Algebra, Automata and Languages

An automaton is a finite state transition system that is used to defined a language. For our purpose, automata is finite state generator of sequences of events. In classical automata theory, usually the transition relation is specified dynamically as a function from the set of states to the power set of the set of states. For example, $T(u, a) = \{v, w\}$ means that at state u, on input a, the possible next states are v or w. The input events may be encodes as a binary vector and hence a may stand for $(0, 1, 1, ..., 0)$ or some thing. However, if we consider this boolean encoding directly, then we can specify the transition relation directly interms of boolean predicates. For example, if the input letters are encoded with 3 boolean variables, x_1, x_2, x_3 and suppose there is a transition from state u to v on 4 input letters whose encodings are $(1, 0, 0), (1, 0, 1), (1, 1, 0), (1, 1, 1)$ then we could rather write the transition as a transition when $x_1 = 1$. Such a coding gives rise to a Boolean algebra of transition predicates. Also, this facilitates a different representation of transition functions as a matrix of Boolean predicates. If there n states in the finite state automaton, then one can have an $n \times n$ matix, such that i, j^{th} entry λ_{ij} is a Boolean predicate that encodes the condition under which the automaton moves from state i to state j.

For those, who know the semantics of synchronous product of two transition systems (A synchronous product can be thought of as a parallel composition where, the two automata are working synchronously in parallel) it is easy to see that the *tensor* product of the two matrices representing two individual transition systems will correspond to the transition system of the synchronous product.

To elaborate, given two automata and its transition system being represented as matrices, the synchronous product is another automaton, whose state set is the cartesian product of the two individual automata. The intuition is that when two automata are working in parallel, the system of the two can be in a state (u, v), where the first one is in state u and the second one is in a state v. Now suppose λ is the transition predicate for the first one to move from state u to u' and let ρ be the transition predicate for the second one to move from state v to v', then the transition predicate for the whole system to move from global state (u, v) to (u', v') is $\lambda * \rho$.

3.2 Some Words about Boolean Algebras

A Boolean algebra is a set L with distinguished elements $0, 1 \in L$, closed under boolean operations $*$(AND), $+$(OR) and $\sim$(NOT) and the properties of the operators are defined interms of a finite set of axioms which we assume that the reader is familiar with.

Now we are going to state a few relevant definitions and well known theorems of Boolean Algebra that are essential in the understanding of the theory developed in the rest of the lectures.

Definition 1 *For $x, y \in L$, we say $x \leq y$ if and only if $x * y = x$. A Boolean algebra is* **complete** *if every set $S \subseteq L$ contains a supremum and infimun with respect to the partial order $\subseteq$.*

$S(L)$ the set of **atoms** *of L, are the nonzero elements of L minimal with respect to the partial order $\leq$.*

A Boolean Algebra is **atomic** *if every nonzero element dominates an atom.*

Fact 1 *Every finite Boolean algebra is atomic. Hence, for our purpose, all Boolean algebra is atomic.*

Fact 2
1. *$s, t \in S(L), s \neq t$ implies $s * t = 0$.*
2. *Every nonzero element of a finite Boolean algebra is a sum of unique set of atoms. (Note that atoms are the minterms)*
3. *The infimum (supremum) of a set of elements is their product (sum).*

Definition 2 *For L, L', two Boolean algebras, a map*

$$\phi : L \rightarrow L'$$

, is a **homomorphism** *provides*

$$\phi(x + y) = \phi(x) + \phi(y)$$

$$\phi(x * y) = \phi(x) * \phi(y)$$
$$\sim \phi(x) = \phi(\sim x)$$

.

The **kernel** *of a homomorphism $\phi : L \rightarrow L'$ is $ker\phi = \{x \in L \mid \phi(x) = 0 \in L'\}$.*

The following lemma is easy to prove and important to remember:

Lemma 1 *Let ϕ be a Boolean algebtra homomorphism. Then the following are true:*

1. $\phi(1) = 1$ *and* $\phi(0) = 0$.
2. $x \leq y$ *implies* $\phi(x) \leq \phi(y)$
3. $ker\phi = 0$ *if and only if* ϕ *is one to one.*

Now we define a few more terms from the area of Boolean algebra and present a key theorem that will be used later.

Definition 3 *A Boolean algebra K is a subalgebra of a Boolean algebra L, if $K \subseteq L$ and they have the same operations and the distinguished elements $O, 1$.*

Definition 4 *For $L_1, L_2, ..., L_k \subseteq L$ subalgebras, we define their product as*

$$\Pi_i L_i = \{\sigma_{j \in J} x_{ij} * ... * x_{kj} \mid x_{ij} \in L_i, J finite\}$$

.

It is easy to see that $\Pi_i L_i$ is a subalgebra of L.

Definition 5 *$L_1, L_2, ..., L_k$ are independent Boolean algebras if $0 \neq x_i \in L_i, i = i, ..., k \Rightarrow x_1 * x_2 * ... * x_k \neq 0$.*

The following lemma , later referred to as *lifting lemma* is a key lemma used in the reduction methodologies. It shows that individual homomorphisms on individual independent components can be lifted to the product. (This is a nice compositionality result).

Lemma 2 *Suppose L, L' are Boolean algebras and $L'_1, L'_2, .., L'_k$ are independent subalgebras of $L' = \Pi L'_i$. If for $i = 1, ..., k$*

$$\phi_i : L'_i \rightarrow L$$

is a homomorphism, then the map

$$\phi : L' \rightarrow L$$

defined by linear extension of

$$\phi(x_1 * ... * x_k) = \phi_1(x_1) * ... * \phi_k(x_k)$$

for $x_i \in L'_i$ is a homomorphism.

Now recall that we have already seen how to represent the transition structure of a finite state automaton a matrix over a boolean algebra. We call such matrices as L matrices. If M be an L matrix over a Boolean algebra L, and the set of states of the automaton is $V(M) = V$, then $M(v,w) = \Sigma_{s \in S(L), s \leq M(v,w)} s$ where each s is the Boolean encoding of an input letter.

$s_M(v) = \Sigma_{w \in V(M)} M(v,w)$ denotes the Boolean predicate corresponding to leaving the state v of the automaton.

Definition 6 *M is* **deterministic** *at state v if*

$$w \neq w' \Rightarrow M(v,w) * M(v,w') = 0$$

. M is deterministic if M is deterministic at each state. M is **complete** *if for all state $v \in V(M), s_M(v) = 1$.*

Definition 7 *Let M, N be $L-$matrices with $V(M) \cap V(N) = \phi$. Their* **direct sum** *$M \oplus N$ is L matrix with $V(M \oplus N) = V(M) \cup V(N)$ and $(M \oplus N)(v,w) = M(v,w) if v, w \in V(M)$, $(M \oplus N)(v,w) = N(v,w) if v, w \in V(N)$, and 0 otherwise.*

Their **tensor product** *$M \otimes N$ is the L-matrix with*

$$V(M \otimes N) = V(M) \times V(N)$$

where

$$(M \otimes N)((v,v'),(w,w')) = M(v,w) * N(v',w')$$

.

3.2.1 Paths and Runs

Definition 8 *A* **path** *in M is a string $\mathbf{v} = v_0, v_1, ..., v_n \in V(M)^{n+1}$ for $n \geq 1$ such that $(v_1, v_{i+1}) \in E(M)$ for $i = 0, ..., n-1$. if $v_n = v_0$, $\mathbf{v}$ is a cycle. $w \in V(M)$ is* **reachable** *from $v \in V(M)$ or $I \subseteq V(M)$ if there is a path $\mathbf{v}$ with $v_0 = v$ or $v_0 \in I$ and $v_n = w$. $C \subseteq V(M)$ is* **strongly connected***, if for all $v, w \in C$, there is a path in C, from v to w.*

Definition 9 *Given a string $(x_0, x_1, ..., x_{n-1}) \in L^*$ (respectively, a sequence $(x_0, x_1, ... \in L^\omega)$ and a string $(v_0, v_1, ..., v_n) \in V(M)^*$ (respectively, a sequence $(v_0, v_1, ... \in L^\omega)$ we say $\mathbf{v}$ is a* **run** *of $\mathbf{x}$ provided for all i,*

$$x_i * M(v_i, v_{i+1}) \neq 0$$

.

3.2.2 Projection or Hiding

Projection corresponds to an abstraction which is more popularly known as *hiding*. In order to abstract from lower level details, it is some times helpful to be able to existentially quantify some variables so that they do not appear in the higher level description.

Definition 10 *Let $L', L'' \subseteq L$, and suppose L', L'' are independent subalgebras of L, with $L = L'.L''$. Let $\Pi_{L'} : L \rightarrow L'$ be the projection given by $\Pi_{L'}(s' * s'') = s'$ for atoms $s' \in S(L'), s'' \in S(L'')$ and extended linearly to L.*

Lemma 3

$$\Pi_{L'}(0) = 0$$
$$\Pi_{L'}(1) = 1$$
$$\Pi_{L'}(x + y) = \Pi_{L'}(x) + \Pi_{L'}(y)$$
$$\Pi_{L'}(x * y) \leq \Pi_{L'}(x) * \Pi_{L'}(y)$$
$$\sim \Pi_{L'}(x) \leq \Pi_{L'}(\sim x)$$

.

3.2.3 L-languages

A **string** is a finite dimensional vector. Let L be a Boolean algebra. An $L - *$ language us a subset of $S(L)^+$, the set of strings of atoms of L, and $L - \omega$ language us a subset of $S(L)^\omega$, the set of infinite sequences of atoms of L.

Definition 11 *Let L, L' be arbitrary Boolean algebras.*

$$\sigma : S(L) \rightarrow S(L')$$

a homomorphism map and define

$$\Phi : S(L)^+ \rightarrow S(L)^+$$

by $\Phi(\mathbf{x}) = (\sigma(x_i))_i$. Then Φ is said to be a **language homomorphism** *with* **support** *σ. Φ naturally extends to ω words and also to subsets of words.*

3.3 L-Automaton and L-Process

System behaviors are defined through finite state generators and acceptors of behavior. If all the behaviors are eventually terminating then the behaviors can be modeled as strings of events. In case of nonterminating behaviors, they have to be modeled by ω-sequences of events.

In the COSPAN approach, systems are modeled as a "behavior generator", which are called **processes**. On the other hand,"behavior acceptor" are **automata**. The acceptor or automaton defined a language whose elements are strings each of which describes a complete path in a complete L-matrix from an intial state to a "final" or "accepting" state. The automata are very much useful in describing *properties*. Dually, the *system* itself is modeled as a generator or *process* such that its language consists of strings that describe path in an incomplete transition matrix from an intial state to a state which is not an **exception** state. Semantically, process and automaton are dual in some sense.

Definition 12 *Let L be a Boolean Algebra. An $L-*$-automaton is a 3-tuple*

$$A = (M_A, I(A), F(A))$$

where M_A is a complete $L-$matrix (the transition matrix), $I(A), F(A) \subseteq V(M_A)$ are initial and final states respectively. The language of A is the $L-*$ language

$$\mathcal{L}(A) = \{\mathbf{x} \in S(L)^+ | \mathbf{x} \text{ has a run in } M_A \text{ from } I(A) \text{ to } F(A)\}$$

.

Definition 13 *An $L-$automaton A is a 4-tuple*

$$A = (M_A, I(A), R(A), Z(A))$$

where

$$M_A \text{ is a complete L-matrix(transition matrix)}$$
$$I(A) \subseteq V(M_A)(\text{ initial states})$$
$$R(A) \subseteq E(M_A)(\text{ recur edges})$$
$$Z(A) \subseteq 2^{V(M_A)}(\text{ cycle sets})$$

Definition 14 *Let M be an $L-$matrix and let $\mathbf{v} \in V(M)^\omega$. Set*

$$\beta(\mathbf{v}) = \{e \in E(M) | (v_i, v_{i+1}) = e \text{ infinitely often}\}$$

.

Definition 15 *For an arbitrary sequence $\mathbf{v}$, set*

$$\mu(\mathbf{v}) = \{v | v_i = v \text{ for infinitely many } v_i's\}$$

Definition 16 *Let A be an $L-$automaton. Then the language of A, $\mathcal{L}(A)$, is the set of $\mathbf{x} \in S(L)^\omega$ such that for some run $\mathbf{v}$ of $\mathbf{x}$ in A with $v_0 \in I(A)$,*

$$\beta(\mathbf{v}) \cap R(A) \neq \phi \text{ or } \mu(\mathbf{v}) \subseteq C \in Z(A)$$

. Such a run $\mathbf{v}$ is called an **accepting** *run of* $\mathbf{x}$.

Definition 17 *An $L - *$-process is a 4-tuple*

$$P = (L_P, M_P, I(P), F(P))$$

where L_P is a subalgebra of L (output subalgebra of P), M_P is an arbitrary $L-$matrix (transition matrix), $I(P), F(P) \subseteq V(M_P)$ (initial and final states). The language of P is the $L - *$ language

$$\mathcal{L}(P) = \{\mathbf{x} \in S(L)^+ | \mathbf{x} \textit{ has a run in } M_P \textit{ from } I(P) \textit{ to } V(P) - F(P)\}$$

Notice that the role of final states in the case of processes is providing exception rather than acceptance.

Definition 18 *An $L-$process is a 5-tuple*

$$P = (L_P, M_P, I(P), R(P), Z(P))$$

where $L_P, M_P, I(P)$ are as in the previous definition. $R(P)$ and $Z(P)$ are the same as in case of L-automaton.

Note 1 *For a process or automaton X, we write $V(X) = V(M_X)$, $|X| = |V(X)|$, $E(X) = E(M_X)$, $X(v, w) = M_X(v, w)$, $s_X(v) = s_{M_X}(v)$ for notational convenience.*

Definition 19 *The selection of an $L-$process P at a state $v \in V(P)$ are the elements of the set*

$$S_P(v) = \{s \in S(L_P) | s * s_P(v) \neq 0)$$

The intuitive meaning of *selection* is a set of nondeterministic outputs as a function of state. The nondeterministic nature of selection provides a powerful method of abstraction.

Definition 20 *Let P be an $L-$process. The language of P is the set $\mathcal{L}(P)$ of $\mathbf{x} \in S(L)^\omega$ such that for some run $\mathbf{v}$ of $\mathbf{x}$ in P with $v_0 \in I(P)$,*

$$\beta\mathbf{v}) \cap R(P) = \phi \textit{ and } \mu(\mathbf{v}) \cap (V(P) \setminus C) \neq \phi \ \forall C \in Z(P)$$

such a run $\mathbf{v}$ is called an **accepting** *run of $\mathbf{x}$.*

Definition 21 *Let P be an $L-$process. The* **dual** *of P denoted as $P^\#$ is an $L-$automaton with ($\# \notin V(P)$)*

$$V(P^\#) = V(P) \cup \{\#\} \textit{ if } M_P \textit{ is incomplete}$$

otherwise $V(P^\#) = V(P)$.

$$P^\#(v, w) = P(v, w) \textit{ if } v, w \in V(P)$$

otherwise for all v

$$P^{\#}(v, \#) = \sim s_P(v)$$

$$P^{\#}(\#, v) = 0$$

$$P^{\#}(\#, \#) = 1$$

$$I(P^{\#}) = I(P)$$

$$Z(P^{\#}) = Z(P)$$

$$R(P) = R(P) \cup (E(P^{\#}) \cap (V(P^{\#}) \times \{\#\})) \ \textit{if} \# \in V(P^{\#})$$

otherwise $R(P^{\#}) = R(P)$.

Lemma 4 $P^{\#}$ *is an* $L-$*automaton.*

Note that one can also define a dual of an L-automaton which will be an L-process. However, since L-automaton is defined with a complete L-matrix, no syntactic change will be needed to define the dual (as was needed while defining the dual of a process).

Lemma 5 *If A is an L-automaton, then* $A^{\#}$ *(interpret the acceptance structure as exception structure), is an L-process.*

Definition 22 *An L-automaton/process X is* deterministic *if* M_X *is deterministic and is strongly deterministic if also* $|I(X)| = 1$.

Lemma 6 *Let X be an L-process or L-automaton. Then* $\overline{\mathcal{L}(X)} \subseteq \mathcal{L}(X^{\#})$. *Moreover, if X is strongly deterministic then*

$$\overline{\mathcal{L}(X)} = \mathcal{L}(X^{\#}).$$

Now we describe some operations on L-automata and L-processes and state some compositionality results in that context.

Definition 23 *Let $X_1, ..., X_k$ be a family of L-automata. Then*

$$\oplus X_i = (\oplus_i M_{X_i}, \cup_i I(X_i), \cup_i R(X_i), \cup_i Z(X_i))$$

$$\otimes X_i = (\otimes_i M_{X_i}, \times_i I(X_i), \cup_i \Pi_i^{-1} R(X_i), \cup_i \Pi_i^{-1} Z(X_i))$$

where $\Pi_i^{-1} Z(X_i) = \{\Pi_i^{-1} C | C \in Z(X_i)\}$. Note that Π_i^{-1} is sort of an inverse for projection function and may be called a cylindrification *function.*

For L-processes we can make an analogous definitions, with respective output subalgebras $L_{\oplus X_i} = L_{X_1}$ (with $\oplus X_i$ being undefined unless $L_{X_1} = ... = L_{X_k}$ and $L \otimes X_i = \Pi L_{X_i}$.

Lemma 7 *Let $A_1, ..., A_k$ be a family of $L-$automata. Then*

$$\mathcal{L}(\oplus A_i) = \mathcal{L}(\otimes A_i) = \cup \mathcal{L}(A_i)$$

Let $P_1, ..., P_k$ be a family of $L-$processes. Then

$$\mathcal{L}(\oplus P_i) = \cup \mathcal{L}(P_i)$$

$$\mathcal{L}(\otimes P_i) = \cap \mathcal{L}(A_i)$$

The above lemma not only gives a nice compositionality result, but also is used heavily in the semantics of the model of coordination in COSPAN.

Now we state a few more definitions and lemmas which are relevant later.

Definition 24 *Let A be a deterministic L-automaton and set $A_i = (M_A, \{i\}, R(A), Z(A))$ for all $i \in I(A)$. Then the* strong determinization *of A is the L-automaton $[A] = \otimes A_i$.*

Lemma 8 *Let A be a deterministic L-automaton. Then $[A]$ is strongly deterministic, and $\mathcal{L}([A]) = \mathcal{L}(A)$.*

Lemma 9 *If P be an L-process with M_P deterministic, then $P_i = (L_P, M_P, \{i\}, R(P), Z(P))$ is strongly deterministic for each $i \in I(P)$ and $\mathcal{L}(\oplus P_i) = \mathcal{L}(P)$.*

Lemma 10 *Given a strongly deterministic L-process P, there exists a strongly deterministic $L-$process Q such that $\mathcal{L}(P) = \mathcal{L}(Q)$ and $|Z(Q)| = 1$.*

Now we just state the definition of homomorphic images of automata or processes induced by Boolean algebra homomorphism.

Definition 25 *For an L-automaton A and a homomorphism $\vartheta : L \to L'$, let*

$$\vartheta A = (\vartheta M_A, I(A), R(A), Z(A))$$

. Similarly one can define homomorphic images of L-processes.

Lemma 11 *Let $\vartheta : L' \to L$ be a homomorphism and let X be an L'-automaton or L'-process. Then*

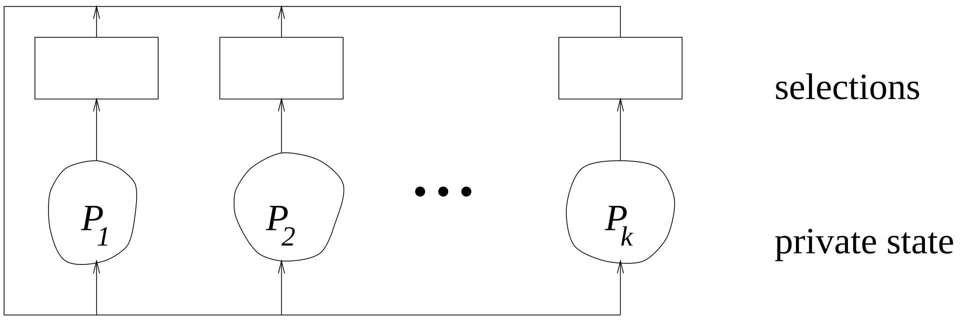
$$\overline{\mathcal{L}(\vartheta X)} \subseteq \mathcal{L}(\vartheta X^{\#})$$

3.4 Selection/Resolution Model

The basic aim of developing all these theory and the verification tool is to analyze the behaviors of real systems. The real systems usually comprise of various components that are interconnected in some fashion and they coordinate with each other in performing specific computational task. Our model of coordination is based on the theory so far developed. The system is modelled with coordinating collection of L-processes and the semantics of their composition is given interms of their synchronous products. The individual processes produce outputs as functions of their inputs and current states and the outputs of some of them serve as input to the others.

Although the model is synchronous, by virtue of nondeterminism it is not difficult to model asynchrony . In fact, realtime systems can also be modeled the same way. However, in this lecture we won't go into the details of that.

We model the system of coordinating processes as a set of L-processes. Each L-process has a set of state variables, which can be seen as its local private memory, and selection variables that are public. The system P consisting of L-processes $P_1, \ldots, P_k$ can be visualized as follows:



The above system functions in a two-phase mode: in *selection phase* all processes write their selection variables based on their local state and then in *resolution phase* every process reads the public memory and updates its private state.

Each process makes a move from its current state based on the values of selection variables and its transition relation. If for every state v, the sum of selections $\Sigma P[v, w] = 1$, the transition relation is complete.

One must note that although this is a synchronous model, it is easy to model asynchronous system on top of this synchronous model by introducing *pause* actions to model arbitrary delays. We introduce an additional requirement that when a process is pausing, it cannot respond to selections of any other process. Also we could simulate an interleaving model by adding a *scheduler* variable. If there are k processes then we add a combinational variable say *sched* which ranges over $\{1, 2, ..., k\}$ and cylindrify over that variable.

So we may comment that the synchronous semantics adopted here is quite general and we can model other semantics efficiently on top of this. That makes the model very attractive.

Definition 26 *L-processes $P_1, ..., P_k$ are* independent *if $L_{P_1}, ..., L_{P_k}$ are independent.*

Coordination in a system of independent L-processes $P_1, ..., P_k$ is modelled by the behavior of the product L-process

$$P = P_1 \otimes P_2 \otimes .... \otimes P_k$$

.

As explained above, one can dynamically interpret this system as a system going through synchronous time steps. In each step (crank) each process P_i makes a selection x_i possible at its current state v_i (in other words $x_i \in S_{P_i}(v_i)$). The selection may be considered as an output of process P_i and can be thought of some thing in its public memory. Other processes P_j takes this output as input (because it is public to all) and then each process resolves as to what state it is going to assume (state is analogous to private memory) which the other processes cannot see.

At this point it must be noted that the language for describing the L-processes and L-automata is also called S/R because the semantics of the language is given interms of the Selection/Resolution just described.

Also, note that L-processes has a natural way of taking care of *fairness* constraints. In other models of coordination, usually fairness constraint needs to be imposed externally. As we know, fairness constraints rule out certain behavioural sequences generated by the model in absence of the constraint. However, the negative acceptance structure of L-processes, rule out behaviours based on the exceptions (cycle sets and recur edges) and hence fairness can be easily treated in the model without externally imposing it.

3.5 Verification based on L-processes and L-automata

As we have already discussed, a system of coordinating processes are modelled by L-processes $P_1, ..., P_k$ and one wants to verify that the system satisfies a list of properties. In other words, one wants to verify that they can perform a list of "tasks". The system model's semantics is given interms of synchronous product of P_is. Let $P = \otimes P_i$. Each task is modelled as an L-automaton T. The L-process P defined the set of all behaviors $\mathcal{L}(P)$ of the modelled system. The L-automaton T defines the set of all behaviots $\mathcal{L}(T)$ consistent with the performance of the modelled task. Now *verification* consists of testing the language containment

$$\mathcal{L}(P) \subseteq \mathcal{L}(T)$$

.

3. Algorithms

Definition 27 *Given L−automaton A, let $A^{\#}$ (dual of A) be A interpreted as an L−process.*

An immediate proposition follows:

Proposition 1 *If A is strongly deterministic then $\mathcal{L}(A^{\#}) = \overline{\mathcal{L}(A)}$.*

Corollary 1 *$\mathcal{L}(P) \subseteq \mathcal{L}(T)$ if and only if $\mathcal{L}(P) \cap \overline{\mathcal{L}(T)} = \emptyset$ where P is an $L-$process and T is a strongly deterministic $L-$automaton representing the task.*

So the problem of verification reduces to checking the emptiness of a product automaton: $\mathcal{L}(P) \bigcap \mathcal{L}(T^{\#}) = \mathcal{L}(P \otimes T^{\#}) = \emptyset$.

Checking the emptiness of an L-process can be done in the following three steps:

1. Compute the reachable subgraph
2. Delete *recur* edges
3. Check if any strongly connected component is **not** contained in some cycle set. This would mean that the language is non-empty.

In practice, the construction is done on-the-fly: a tree of reachable states is constructed. Each time a *recur* edge is crossed, a new tree is started. Testing back edges that are encountered along the way allows one to identify bad cycles early.

So the complexity is $O(\mid E(P) \mid\mid Z(P) \mid)$. If the size of P is acceptably small then this is small but as we know that the model checking problem itself is PSPACE-complete, so the best we could do is to develop efficient heuristics.

State Explosion problem

Let $P = \otimes P_i$. A straight forward algorithm will build P to test if $\mathcal{L}(\otimes P_i) \subseteq \mathcal{L}(T)$. But if $\mid P_i \mid$ is c then $\mid P \mid = c^k$ which is exponential in the size of the description. This problem is referred to as the state explosion problem.

Note that if the verification is done with explicit state space, then state explosion occurs. However, there are other implicit representation of states (like Ordered Binary Decision Diagrams which is not discussed here) the term "state explosion" does not convey the complexity theoretic hardness in the proper way.

One useful heuristic is to utilize modularity of $P = \bigotimes P_i$. For every component, $\mathcal{L}(\otimes P_i) \subseteq \mathcal{L}(P_i)$. Thus is we are able to prove $\mathcal{L}(P_i) \subseteq \mathcal{L}(T)$ for some P_i, it would imply the desired result.

Space-Time Trade Off

Space efficient algorithms can be designed based on s-t connectivity problem by mimicking Savitch's construction. That way it is possible to get $O(k^2)$ space and $2^{\Omega(k^2)}$ time algorithm. But this is too intolerable time bound. So we will now describe four different heuristics for reducing the space bound with better time bounds.

Four such methods will be discussed now. They are :

- Localization
- Task Decomposition
- Reduction Hierarchy
- Homomorphic Reduction

4 Reduction Methodologies

Localization Reduction
Suppose we want to verify

$$\mathcal{L}(P_1 \otimes .... \otimes P_k) \subseteq \mathcal{L}(T)$$

for some task T but in fact, there is some $J < k$ such that

$$\mathcal{L}(P_{i_1} \otimes .... \otimes P_{i_j}) \subseteq \mathcal{L}(T)$$

then the original containment check succeeds for a less computational expense. However, how to find this subset of processes for which the property T is **local**. In other words, if the above situation occurs, then the for proving that property T holds, the other processes are irrelevant and hence can be dispensed with. But one has to find out which ones.

Depending on the property being verified, some details of the system description may be irrelevant for verification purposes. When a concurrent system is described as a product automata, some of the component automata are irrelevant in some sense for the verification purpose. More generally, if the property T is a local property for some part of the system then localization may be used to abstract away the irrelevant part.

Localization algorithm works as follows: Consider a variable dependency graph showing how values of each variable depends on the values of other variables in the resolution stage. We start with the variables that T depends on. Arrange the rest of the graph in levels by their distance from the selected variables. The variables on the next closest level are "free"ed, that is, have non-deterministic values assigned to them. It can be easily seen that by freeing variables one gets a larger language. Thus, if we can prove the desired property on this abstracted system, it is also true of the original system. However, now the negative result does not necessarily mean that the original system is incorrect: we may have abstracted away some relevant variables. The verification routine produces a counterexample (an execution trace of the abstracted system) that can be analyzed. It may lead to an error in the specification, or it may turn out to be spurious. In the latter case, we use the trace to decide, which variables should be "unfree"d on the next level of the dependency graph.

The above process can be repeated with the new set of variables. We thus expand the state space iteratively until either we are able to prove the desired property, or find an error, or run out of space. The method can be expected to work when the property is indeed local.

Task Decomposition
If we could decompose the given specification or task T into subtasks T_i such that each T_i is a local property and $\bigcap_i \mathcal{L}(T_i) \subseteq \mathcal{L}(T)$, then we could use localization heuristic to circumvent the state explosion problem in a hierarchical fashion. However, there is no automated method for task decomposition. One has to have enough intuition about the design and the task to get a reasonable decomposition. So the question is *How to decompose a task?*.

Reduction Hierarchy

A task T is decomposed into several subtasks $T_{11}, T_{12}, ..., T_{1k}$ and each of them are decomposed into their subtasks. Such a decomposition can be depicted as a tree (or a DAG). This hierarchy is called a reduction hierarchy if the following condition holds for every decomposed task X which has been decomposed into Y, Z(say). The condition is $\mathcal{L}(Y) \cap \mathcal{L}(Z) \subseteq \mathcal{L}(X)$.

To make the decomposition worth while, we have to make sure that the size of each component on all levels, including leaves, is comparable to the size of the original property. Also, the outdegree of the decomposition tree should be bounded by a constant.

It is clear that if we have a reduction hierarchy then we have a task decomposition. But interesting questions are :

- Does there always exist a reduction hierarchy for arbitrary tasks?
- Is there an efficient algorithm (polynomial in the number of components in the original system) to find an efficient reduction hierarchy when one exists?

Homomorphic Reductions

This is an abstraction mechanism that works on the underlying boolean algebra. In certain cases, we can find a boolean algebra homomorphism $\phi : L' \rightarrow L$ that will induce a language homomorphism between L-processes and L'-processes such that the following property will hold:

$$\begin{gathered}\mathcal{L}(P) \subseteq \mathcal{L}(T) \\ \Uparrow \\ \mathcal{L}(P') \subseteq \mathcal{L}(T')\end{gathered}$$

Here, P and P' are L- and L'-processes, respectively, and T and T' are L- and L'-automata. If the language homomorphism is such that P' is significantly less than P, and T' is not significantly larger than T, we have reduced our problem to an easier one.

An example of a homomorphic reduction is data abstraction.

In fact the localization reduction is a special case of homomorphic reduction. However, localization can be done algorithmically where as homomorphic reduction needs to be guessed.

Definition 28 *Let P be an $L-$process and T be an $L-$automaton and let P' be an L' process and T' be an L'-automaton. Let $\Phi : S(L)^\omega \rightarrow S(L')^\omega$ be a language homomorphism staisfying*

$$\Phi\mathcal{L}(P) \subseteq \mathcal{L}(P'), \Phi\mathcal{L}(T^\#) \subseteq \mathcal{L}(T'^\#)$$

, then Φ is said to be **co-linear** *for $(P, T; P', T')$.*

Moreover, if

$$\Phi\mathcal{L}(P) = \mathcal{L}(P')$$

, then Φ is called **strongly** *co-linear for $(P, T; P', T')$.*

Theorem 1 *Let Φ be a language homomorphism co-linear for $(P, T; P', T')$ and suppose T' is strongly deterministic. Then*

$$\mathcal{L}(P') \subseteq \mathcal{L}(T') \Rightarrow \mathcal{L}(P) \subseteq \mathcal{L}(T).$$

Moreover, if Φ is strongly co-linear then

$$\mathcal{L}(P') \subseteq \mathcal{L}(T') \equiv \mathcal{L}(P) \subseteq \mathcal{L}(T).$$

If for strongly deterministic T' there exists a homomorphism that is co-linear for $(P, T; P', T')$, we say that (P', T') is a **homomorphic reduction** of (P, T).

Homomorphic reduction can be very useful in rendering the verification problem tractable. In some cases, even parameterized models can be verified for all possible values of the parameters if there is a suitable homomorphic reduction.

There are methods to check the co-linearity algorithmically once the reduction has been guessed.

For a global task T, it may not be easy to find a sufficiently simple reduction (P', T') from (P, T) such that $|V(P')|$ is small enough. However, if one can guess a task decomposition $T_1, T_2, ..., T_n$ such that $\cap_i \mathcal{L}(T_i) \subseteq \mathcal{L}(T)$ and such that each (P, T_i) has much simpler reductions. It is often a good strategy to find a good decomposition $T_1, T_2, ..., T_n$ of the given task and find reductions (P_i', T_i') for (P, T_i) and verify that $\mathcal{L}(P_i') \subseteq \mathcal{L}(T_i')$ and this conclude that $\mathcal{L}(P) \subseteq \mathcal{L}(T_i), \forall i$ which would imply that

$$\mathcal{L}(P) \subseteq \cap \mathcal{L}(T_i) \subseteq \mathcal{L}(T)$$

Experience shows that decomposing T might be easy in some cases.

We should insert some methods for verifying homomorphism and co-linearity at this point

4.1 Homomorphic Reduction and Refinement based Topdown methodology

Homomorphic reduction gives a methodology for a stepwise refinement based design methodology. Before we describe how that is accomplished, we need to show the correspondence between Boolean algebra homomorphism and language homomorphism.

Recalling Boolean Algebra Homomorphism

Definition 29 *Let L be a Boolean algebra. Let $S(L)$ be the set of* **atoms** *of L. $S(L)$ is the alphabet over which we define our automata.*

A typical example of an atom is [3] $\{(x_1 = a_1) * (x_2 = a_2) * ... * (x_n = a_n)\}$.

[3] $*$ denotes logical AND

We define a partial order "$\leq$ over L as follows:

$$\forall x, y \in L \; x \leq y \equiv x * y = x$$

.

Definition 30 *Let L, L' be arbitrary boolean algebras. Let*

$$\sigma : S(L) \rightarrow S(L')$$

be a Boolean algebra homomorphisms. Then

$$\Phi : \{S(L)\}^{\omega} \rightarrow \{S(L)\}^{\omega}$$

is a language homomorphism with support σ if it is defined pointwise by σ. In other words if

$$\Phi((x_1, x_2, ..., x_n)) = (\sigma(x_1), \sigma(x_2), ..., \sigma(x_n))$$

.

In homomorphism checking, we have to check if the language of the abstract model is homomorphic to the language of the original model and we need an algorithmic method to do so. Here, we discuss, the duality of the language homomorphism and the Boolean algebra homomorphism and show that for each language homomorphism there is a unique Boolean algebra homomorphism and vice versa. As a result, checking for language homomorphism can be reduced to checking Boolean algebra homomorphism. Hence when the user guesses the Boolean algebra homomorphism, it can be automatically checked if that leads to the corresponding language homomorphism.

Definition 31 *Let $\phi : L' \rightarrow L$ be a Boolean algebra homomorphism. Its support is $\hat{\phi} : L \rightarrow L'$ such that*

$$\hat{\phi}(x) = \Pi_y \{y \in L' \mid x \leq \phi(y)\}$$

.

Lemma 12

$$\hat{\phi}(S(L)) \subseteq S(L')$$

.

Theorem 2 *Over* **finite** *Boolean algebras there is a one to one relationship between language homomorphisms and Boolean algebra homomorphisms.*

Let ϕ_σ be the language homomorphism with support σ. There exists a unique Boolean algebra ϕ with

$$\hat{\phi} \mid_{S(L)} = \sigma$$

. In the reverse direction, if ϕ is a Boolean algebra homomorphism then there is $\phi_{\hat{\phi}|_{S(L)}}$ which is a language homomorphism.

Let

$$\phi : L' \to L$$

. Let P' be an $L'-$process. then $\phi P'$ is an $L-$process such that

$$(\phi P')(v, w) = \phi(P'(v, w))$$

. Let $P = \otimes P_i$ and $P' = \otimes P_j^{'}$. Let Φ denote $\phi_{\hat{\phi}|_{S(L)}}$ from the previous theorem.

Lemma 13

$$\phi(\otimes P_j^{'}) = \otimes \phi(P_j^{'}$$

.

Corollary 2

$$\Phi(\mathcal{L}(\otimes P_i)) \subseteq \mathcal{L}(\otimes P_j^{'}) \;\equiv\; \mathcal{L}(\otimes P_i) \subseteq \mathcal{L}(\otimes \phi P_j^{'})$$

Theorem 3 *Let $P_1, P_2, ..., P_m$ be a set of processes, I_j is a set of index sets such that $\cup I_j = \{1, 2, .., m\}$. If for all j,*

$$\mathcal{L}(\otimes_{i \in I_j} P_i) \subseteq \mathcal{L}(\phi P_j^{'})$$

then

$$\Phi(\mathcal{L}(\otimes_{i \in I_j} P_i)) \subseteq \mathcal{L}(P_j^{'})$$

This gives us the methodology for componentwise verification in some sense.

Examples are

- data abstraction
- abstraction
- Symmetry based reduction

Top-Down Design Methodology

Homomorphic Refinement:

Homomorphic reduction provides a mechanism for a top-down development methodology based on "stepwise refinement". In this methodology, a succession of models are generated. From an abstract specification to detailed implementation. Each model is taken as a "specification" of the next model and the verification is used to check the conformance of the new implementation to the latest specification. This can be done by establishing the homomorphism between the successive levels. This way, the propert that any task whose performance is proved at one level of abstraction is guranteed to be performed at the subsequent levels also. This allows us to verify properties at the simplest model in which it can be defined.

Successive refinement has another great advantage that it allows us to do early debugging before more detailed design has been constructed.

Given two system models at successive levels of details, a more abstract L'-process P' and a more detailed $L-$process P, we verify the refinement of P' to P, by constructing a homomorphism $\vartheta : L' \rightarrow L$ such that the language homomorphism Φ with support $\hat{\vartheta}|_{S(L)}$ satisfies

$$\Phi\mathcal{L}(P) \subset \mathcal{L}(P')$$

.

We say that P is a *refinement* of P' if there is such a Φ. If P is a refinement of P', then for any strongly deterministic $L'-$ automaton T' for which $\mathcal{L}(P') \subseteq \mathcal{L}(T')$, it follows that $\mathcal{L}(P) \subseteq \mathcal{L}(\vartheta T')$ because we have already seen from previous discussion on *homomorphic Images* of automata that $\Phi\mathcal{L}(\vartheta T'^{\#}) \subseteq \mathcal{L}(T'^{\#})$. This shows that any task that is guaranteed to hold for the abstract model P' holds for the more detailed model P provided P is a refinement of P' in the technical sense described above.

Ideally the last model in the refinement hierarchy will be detailed enough so that one can compile its description into software or some hardware layout as the case may be.

4.2 Inductive Abstraction

This is to verify properties of arbitrary composition of similar processes. In the literature, they have been called parameterized systems also. For example, Suppose there is a symmetric or uniform solution to the dining philosophers problem (As has been shown in Dining Philosophers with Encyclopedia). If one wants to show its liveness and safety properties for n philosophers, for arbitrary n, then one can use this inductive method.

This method is not fully automatic in the sense that an "invariant" process must be provided to carry out an induction step. Given such a process, however, the checking procedure reduced to an ordinary language containment check for $L-$processes of fixed size.

Often the invariant process may be obtained by composing a small number of system processes, then making minor modifications. In other cases, no modifications might be necessary. In some other cases, a simple invariant process may be obtained in an *ad hoc* fashion.

With an invariant constructed, there are three steps for proving correctness of parameterized systems. First is ofcourse the base case. The invariant must have all the behaviors of the product of some finite m number if system processes. Second is the induction step, the product of the invariant with the $(m+1)^{th}$ system process is generalized by the invariant. Thirdly, satisfaction test: the invariant process must perform the task.

All the three steps can be carried out automatically with COSPAN. So one can guess the invariant several times until a correct one is obtained and test each time with COSPAN all the three steps.

This is comparable to coming up with the strongest induction hypothesis in inductive proofs in mathematics.

The methodology is based on the following theorem.

Let $\mathcal{P}$ be a ser of L-processes and let $\phi : L \to L'$ be a Boolean Algebra homomorphism which for any $P \in \mathcal{P}$, satisfies $\phi P \in \mathcal{P}$. Thus, ϕ induces a map from $\mathcal{P}$ to $\mathcal{P}$.

Theorem 4 *Let $\{P_1, P_2, , ..., P_n\}$ and $\{Q_1, Q_2, ..., Q_n\}$ be two sets of processes in $\mathcal{P}$. Let m be an integer $1 \leq m < n$ such that for all i satisfying $m < i < n$, $P_{i+1} = \phi(P_i)$ and for all i satisfying $m \leq i < n$, $Q_{i+1} = \phi(Q_i)$. If*

$$\mathcal{L}(\otimes_{i=1}^{m} P_i) \subseteq \mathcal{L}(Q_m)$$

and

$$\mathcal{L}(Q_m \otimes P_{m+1})\mathcal{L}(Q_{m+1})$$

then for all k satisfying $m \leq k \leq n$,

$$\mathcal{L}(\otimes_{i=1}^{k} P_i) \subseteq \mathcal{L}(Q_k)$$