High-Level Execution Time Analysis

Farn Wang[1]

Institute of Information Science, Academia Sinica

Nankang, Taipei, Taiwan 115, R.O.C.

+886-2-7883799 ext. 1717 FAX : +886-2-7824814 farn@iis.sinica.edu.tw

[1]Supported by National Science Council, Taiwan, ROC under grant NSC 86-2213-E-001-009

High-Level Execution Time Analysis

Farn Wang[2]

Institute of Information Science, Academia Sinica

Nankang, Taipei, Taiwan 115, R.O.C.

+886-2-7883799 ext. 1717 FAX : +886-2-7824814 farn@iis.sinica.edu.tw

[2]Supported by National Science Council, Taiwan, ROC under grant NSC 86-2213-E-001-009

Abstract

A compositional algebra, called CAN, for the execution time analysis of high-level software processes is introduced. In CAN, processes with Boolean parameters can be concatenated, concurrently executed, and recursively invoked. We show that the set of execution times of a CAN is semilinear. We then propose and analyze an algorithm which calcuates the execution time sets of a CAN in semilinear forms. Finally, we consider several interesting variations of CAN whose execution time sets can be computed with algorithms.

1. α (alpha) ..... process name subscript
2. β (beta) ..... process name subscript
3. χ (chi) ..... Boolean sequence superscript
4. Δ (capital delta) ..... edge-weighted graph
5. ϵ (epsilon) ..... empty process
6. η (eta) ..... arity function of process names
7. γ (gamma) ..... process name subscript
8. μ (mu) ..... labeling function of trees
9. ν_P (nu) ..... variable used in determining the nonzero executability
10. Π (capital pi) ..... set of process names
11. Ψ (capital psi) ..... Parikh's mapping
12. σ (sigma) ..... a symbol in an alphabet
13. Σ (capital sigma) ..... alphabet
14. τ (tau) ..... semilinear expressions
15. θ (theta) ..... a rule
16. Θ (capital theta) ..... set of rules
17. A ..... a compositional algebra of numbers (CAN)
18. $[\![A]\!]$ ..... semantics, set of process execution times
19. B ..... a bone tree
20. C ..... a contributing tree
21. D ..... a compositional algebra of numbers (CAN)
22. E ..... set of arcs in a tree
23. F ..... a compositional algebra of numbers (CAN)
24. H ..... a subtree
25. $\mathcal{I}$ (calligraphical capital I) ..... an interpretation
26. K ..... a subtree
27. L ..... a language
28. L_A ..... twice the biggest constant used in A
29. M ..... mode in statechart or modechart
30. M_A ..... maximum of minimum execution times of all process types
31. $\mathcal{N}$ (calligraphical capital N) ..... set of natural numbers
32. P ..... process name
33. $[\![P]\!]_A$ ..... set of process execution times
34. Q ..... process name
35. R ..... process name
36. $\mathcal{R}^+$ (calligraphical capital R) ..... set of nonnegative real numbers

37. S ..... starting process symbol
38. T ..... an execution tree, a derivation tree
39. U ..... set of nodes in an edge-weighted graph
40. V ..... set of nodes in a trees
41. W ..... set of arcs in an edge-weighted graph
42. X ..... parameter sequence
43. Y ..... parameter sequence
44. $\mathcal{Z}$ (calligraphical capital Z) ..... set of integers
45. $a, b, c, d, h, i, j, k, l, m, n$ ..... nonnegative integers
46. r ..... root in a tree
47. u, v ..... nodes in a tree or a graph
48. w ..... a string in a language
49. x, y, z ..... parameter variables

High-Level Execution Time Analysis

Farn Wang[3]

Institute of Information Science, Academia Sinica

Nankang, Taipei, Taiwan 115, R.O.C.

+886-2-7883799 ext. 1717 FAX : +886-2-7824814 farn@iis.sinica.edu.tw

Keywords : real-time, specification, verification, software recursion, concurrency, parameter passing, context-free, semilinear

1 Introduction

In the design of sophisticated systems, usually various nonregular behaviours are described and need to be verified. Classic nonregular behaviors refer to those can not be described by finite state automata and are often observed in systems implemented with unrestricted software recursion, multiple infinite queues, range-unbounded variables, ... etc. Although the research of formal methods and automated verification are booming with success to some extent reported elsewhere, in general there has still been very little we can do in analyzing such sophisticated systems. For example, in the model-checking approach of real-time systems, it is usually difficult to determine the timing constants corresponding to the true timing behaviors of program codes. It is our belief that *by adopting certain specification paradigms, some real-time systems, with highly nonregular behavior descriptions in the classic specification theories, actually exhibit regularities that allow mechanical analysis and verification.*

In this paper, we propose a new theory for the execution time analysis of high-level behaviors. We call the theory the *Compositional Algebra of Numbers (CAN)*, and show that it helps identify the behavioral regularity of systems which may have previously been thought to be nonregular. Intuitively, numerical addition represents the execution time of two concatenated sequential processes. For example, we may have a process construction rule $S\langle x, y\rangle \rightarrow P\langle x\rangle Q\langle 1, x, y\rangle$ which means process S with Boolean parameters x, y can be completed by first invoking a process P with parameter x and after the completion of $P\langle x\rangle$, invoking a process Q with parameters $1, x, y$. Note here x, y are unknown Boolean parameters at the time of the rule specification.

[3]Supported by National Science Council, Taiwan, ROC under grant NSC 86-2213-E-001-009

Parallel executions in CAN are modeled by the minimum and maximum selection operations. For example, we may have another rule $S\langle x, y\rangle \rightarrow \min(P\langle x\rangle, Q\langle 1, x, y\rangle)$ which means process S with Boolean parameters x, y is invoked by the simultaneous invocation of processes P with parameter x and Q with parameters $1, x, y$ and is fulfilled when either one of $P\langle x\rangle$ and $Q\langle 1, x, y\rangle$ is fulfilled. Thus the execution time of $S\langle x, y\rangle$ can be defined as the smaller of those of $P\langle x\rangle$ and $Q\langle 1, x, y\rangle$. Examples of such parallel execution include the parallel search in a distributed database, ... etc.

Similarly, a rule like $S\langle x, y\rangle \rightarrow \max(P\langle x\rangle, Q\langle 1, x, y\rangle)$ means process $S\langle x, y\rangle$ is fulfilled when both process $P\langle x\rangle$ and $Q\langle 1, x, y\rangle$ have been fulfilled. Thus the execution time of $S\langle x, y\rangle$ can be defined as the bigger of those of $P\langle x\rangle$ and $Q\langle 1, x, y\rangle$. Examples of such parallel execution include the parallel mergesort, ... etc.

Example 1 *:* Suppose we have a CAN with process types S, P, Q, starting process S with parameters $1, 1$, and process construction rules :

$$
\begin{array}{l}
S\langle x, 0\rangle \rightarrow (3); \\
S\langle x, y\rangle \rightarrow S\langle x, 0\rangle S\langle 0, y\rangle; \\
S\langle x, y\rangle \rightarrow P\langle x\rangle Q\langle 1, x, y\rangle; \\
S\langle x, y\rangle \rightarrow \min(Q\langle x, y, 1\rangle, S\langle y, 0\rangle); \\
S\langle x, y\rangle \rightarrow \max(P\langle x\rangle, S\langle 0, y\rangle); \\
P\langle x\rangle \rightarrow (4); \\
Q\langle x, y, 1\rangle \rightarrow (6)
\end{array}
$$

The first rule says that an $S\langle x, 0\rangle$ process (S process with parameters $x, 0$) can take 3 time units to complete. The next four rules say that an $S\langle x, y\rangle$ process can be the concatenation of an $S\langle x, 0\rangle$ process and an $S\langle 0, y\rangle$ process, or be the concatenation of a $P\langle x\rangle$ process and a $Q\langle 1, x, y\rangle$ process, or be the parallel execution of a $Q\langle x, y, 1\rangle$ and an $S\langle y, 0\rangle$ process (fulfilled when either $Q\langle x, y, 1\rangle$ or the child $S\langle y, 0\rangle$ is fulfilled), or be the parallel execution of a $P\langle x\rangle$ and an $S\langle 0, y\rangle$ processes (fulfilled when both $P\langle x\rangle$ and the child $S\langle 0, y\rangle$ are fulfilled). ||

Given a CAN A with a starting process type with Boolean constant parameters, a fundamental problem is

"What is the set of execution times generated by A (written as $[\![A]\!]$) ?"

It will be good if we can compute a finite representation which describes the set. We shall show that for every CAN A, $[\![A]\!]$ is semilinear [4]. In section 2, we shall compare with related work. In section 3, we

[4] A set of integers is seminlinear iff it can be represented as the union of a finite number of sets like $\{a + b_1 j_1 + \ldots + b_n j_n \mid \forall 1 \leq i \leq n(j_i \in \mathcal{N})\}$ for some $a, b_1, \ldots, b_n \in \mathcal{N}$. Semilinear integer sets are closed under intersection, union, and complement and are subject to efficient manipulation.

shall formally define CAN. The usage of CAN in helping the verification of high-level software systems can be illustrated by the examples in subsections 3.3, 3.4, and 3.5. In section 4, we shall derive an algorithm to compute the semilinear description of any given CAN.

Finally, in section 5, we discuss several other possibilities in using the numerical properties of high-level behaviors. The first one is called *PCAN (Parallel CAN)* which adopts a rendezvous semantics for the parallel operators while disallows arbitrary concatenations. The second one, called *SCAN (Stratified CAN)*, allows finite alternation of concatenation and parallel rendezvous. The third one, called *DRCAN (Deterministic Recurrence CAN)*, allows integer parameter sharing between child processes and connects to the theory of recurrence equations.

1.1 Notations

Notationally, we let $\mathcal{N}$ be the set of nonnegative integers. Given a set H, we let $|H|$ be the size of H. Given a sequence H, we let $|H|$ be the length of H.

Given a rule θ like "$P\langle X\rangle \rightarrow \ldots\ldots$;" where X is a sequence of Boolean parameters, we shall let $\mathrm{lhs}(\theta) = P\langle X\rangle$, i.e. the left-hand-side of rule θ is $P\langle X\rangle$.

A path is a directed graph in which all nodes form a single line. The length of a path is the number of arcs (edges) in the line.

A *tree* is a directed acyclic graph such that there is a special node called *root* and from the root to every other node in the tree, there is exactly one directed path. A node with no outgoing arcs is *external* and is called a *leaf*. A node with outgoing arcs is called *internal*. The *height* of a tree is the length of the longest path in the tree.

Finally, regarding the manipulation of semilinear expressions used in this paper, some discussions can be found in [17]. We accept the following notation. Given a linear integer set $\{a + b_1 j_1 + \ldots + b_n j_n \mid j_1, \ldots, j_n \in \mathcal{N}\}$, we shall denote it as $a + b_1 * + \ldots + b_n *$. A semilinear expression is a union of finite number of sets like $a + b_1 * + \ldots + b_n *$. In particular, it can be shown that for all $a, b_1, b_2 \in \mathcal{N}$,

$$\begin{array}{rll} a + b_1 * + b_2 * = & & \{a + b_1 j_1 + b_2 j_2 \mid j_1, j_2 \in \mathcal{N}; b_1 j_1 + b_2 j_2 < \mathrm{lcm}(b_1, b_2)\} \\ & \cup & (a + \mathrm{lcm}(b_1, b_2) + \gcd(b_1, b_2)*) \end{array}$$

With such an equivalence relation, we can translate all semilinear expressions into unions of sets like $a + b*$. Then it can also be shown that the intersection of two sets, $a_1 + b_1*$ and $a_2 + b_2*$, is empty iff $a_1 - a_2$ is not a multiple of $\gcd(b_1, b_2)$.

Suppose we are given semilinear expressions τ_1, τ_2. We can define the addition of τ_1 and τ_2, in symbol $\tau_1 + \tau_2$, as $\{a_1 + a_2 \mid a_1 \in \tau_1; a_2 \in \tau_2\}$. We can also define the Kleene closure τ_1* of τ_1 as the

minimum set τ that satisfies the following condition : $\tau_1 \subseteq \tau \wedge \forall a \in \tau \forall b \in \tau (a + b \in \tau)$. Some other interesting properties of semilinear expressions are $(a + b_1 * + \ldots + b_n *)* = a * + b_1 * + \ldots + b_n *$ and $(\tau_1 \cup \ldots \cup \tau_m)* = \tau_1 * + \ldots + \tau_m *$ where $\tau_1, \ldots, \tau_m$ are arbitrary semilinear expressions.

2 Related work

In the computation of concurrent systems, there are two major operations that control the system behavior : communication and time passage[10, 14]. In the traditional theory of formal verification[1, 3, 5, 6, 10, 11, 16, 17], for example, temporal logics, process algebra, and first-order logics, universal communication operators, e.g. broadcasting, are very often adopted and many good theoretical and application results have been reported in recent years [2, 4]. Still very little has been done to exploit specification structures and engineering designing rules to alleviate the burden of verification. In contrast, in a CAN system description, the system is defined as a *numerical process* which in turn may be recursively constructed from other numerical processes. The processes are called *numerical* because from outside, we can only observe their full execution times. Thus to some extent, CAN paradigm matches the concept of abstraction and information hiding in a uniform way. The communication among child numerical processes is restricted to the construction of their parent while the communication inside child numerical processes is kept invisible to the parent.

The classic work of R.J. Parikh on context-free language[15] can be reinterpreted as a special case of our CAN. Assume we are given a language L with finite alphabet Σ in sequence $\sigma_1 \sigma_2 \ldots \sigma_n$. If there is a string $w \in L$, then *Parikh's mapping* of w, denoted as $\Psi[w]$, is a vector $(c_1, c_2, \ldots, c_n)$ such that for each $1 \leq i \leq n$, c_i is the number of σ_i occurrences in w. Also $\Psi[L] = \{\Psi[w] \mid w \in L\}$.

Parikh's fundamental work shows that given a context-free language L, $\Psi[L]$ is semilinear. We should point out that Parikh's result talks about semilinearity of integer vector sets. In our framework, we only care about the case of unary alphabet. Given $w \in L$ with unary alphabet, we interpret w as a particular computation of a process type and $\Psi[w]$ as the computation time of process w.

While CAN is not context-free, Parikh's work in the unary alphabet case syntactically corresponds to the *BPA (Basic Process Algebra)* of classical process algebra and represents a truly context-free fragment of CAN since it lacks the parallel operator. Thus we also give Parikh's subclass of CAN the name of *Basic CAN (BCAN)*. We envision the possibility of extending Parikh's work with parallel operators to enhance our ability in specifying and verifying the timing aspects of nonregular real-time systems.

As compared to the classic work of context-free language over general alphabets[7], we see our work

may lead to more verifiable real-time concurrent system specification. For one thing, in the general case, the emptiness problem of classical context-free language intersection is undecidable[8] while in section 4 and 5, we shall see that under the CAN and SCAN paradigms, the problem becomes decidable and forms a plausible framework for real-time system specification.

CAN does bear similar syntactical structure to classic process algebra theory[3, 5]. Thus it is possible to treat CAN as a new semantic definition of process algebra. However, in classic process algebra, parallel composition is defined on interleaving semantics and there is really nothing similar to our parallel operator. But we feel that our parallel composition semantics for CAN is adequate for the abstract description of concurrency in complex real-time systems.

3 CAN

3.1 Syntax

A CAN A is a tuple $\langle \Pi, \eta, S\langle X\rangle, \Theta\rangle$ where Π is the set of process names, η defines the number of parameters received by each process, $S\langle X\rangle$ is the starting process and Boolean parameter values, and Θ is the set of process construction rules. For each $P \in \Pi$, $\eta(P)$ is called the *arity* of the process type. The rules in Θ are in one of the following four forms.

$$\begin{array}{lcl} P\langle X\rangle & \rightarrow & (c); \\ P\langle X\rangle & \rightarrow & P_1\langle X_1\rangle P_2\langle X_2\rangle; \\ P\langle X\rangle & \rightarrow & \max(P_1\langle X_1\rangle, P_2\langle X_2\rangle); \\ P\langle X\rangle & \rightarrow & \min(P_1\langle X_1\rangle, P_2\langle X_2\rangle); \end{array}$$

Here X is a sequence $x_1, \ldots, x_n$ of Boolean parameters which can be either Boolean variables or Boolean constants. c is an integer constant in $\mathcal{N}$ and P, P_1, P_2 are process types in Π. When $\eta(P) = 0$, we may also abbreviate $P\langle\rangle$ as P.

The intuition behind each rule is the following.

- Rule $P\langle X\rangle \rightarrow (c)$; means that process P with parameters X can be executed with c time units.
- Rule $P\langle X\rangle \rightarrow P_1\langle X_1\rangle P_2\langle X_2\rangle$; means that process P with parameter X can be executed by first executing a process P_1 with parameters X_1 and then after the completion of $P_1\langle X_1\rangle$, immediately executing a process P_2 with parameters X_2.
- Rule $P\langle X\rangle \rightarrow \min(P_1\langle X_1\rangle, P_2\langle X_2\rangle)$; means that process P with parameters X can be executed by simultaneously executing processes P_1 with parameters X_1 and P_2 with parameters X_2 and the execution is completed when either $P_1\langle X_1\rangle$ or $P_2\langle X_2\rangle$ is completed.

- Rule $P\langle X\rangle \rightarrow \max(P_1\langle X_1\rangle, P_2\langle X_2\rangle)$; means that process P with parameter X can be executed by simultaneously executing processes P_1 with parameters X_1 and P_2 with parameters X_2 and the execution is completed when both $P_1\langle X_1\rangle$ and $P_2\langle X_2\rangle$ are completed.

We require that in any given CAN $\langle \Pi, \eta, S\langle X\rangle, \Theta\rangle$, for each $P \in \Pi$, if $P\langle X\rangle$ is used in Θ, then $|X| = \eta(P)$. That is for each process invocation, its number of parameters must be consistent with its process' arity.

Although the syntax of our composition rules is restricted to exactly two child processes, we note the readers that this is purely for convenience of presentation. In practice, we may want to extend the rules to incorporate rules with other than two child processes. For example, in section 4, sometimes we write rules with one child process ($P \rightarrow Q$). The definitions and proofs in the paper can be modified to accomodate such extension without any difficulty.

3.2 Semantics

Suppose we are given a CAN $A = \langle \Pi, \eta, S\langle X\rangle, \Theta\rangle$. An *interpretation* for A is a mapping from the set of free Boolean parameter variables used in A to the set of Boolean constants. Given an interpretation $\mathcal{I}$ and a sequence X of Boolean variables and Boolean constants, $\mathcal{I}(X)$ is a sequence of Boolean constants obtained from X by for every free variable $x \in X$, substituting x for $\mathcal{I}(x)$. Given a rule $\theta \in \Theta$ and an interpretation $\mathcal{I}$, we let $\theta^{\mathcal{I}}$ be the *instantiated rule* obtained from θ by substituting every free variable x in θ for $\mathcal{I}(x)$. We let $\Theta^{\mathcal{I}}$ be the set of instantiated rules in Θ with respect to $\mathcal{I}$, that is, $\Theta^{\mathcal{I}} = \{\theta^{\mathcal{I}} \mid \theta \in \Theta\}$. Also, we let $\Theta^{\forall}$ be $\bigcup_{(\mathcal{I} \text{ is an interpretation for } A)} \Theta^{\mathcal{I}}$.

We can also modify the derivation trees of context-free languages to formally define the semantics of CAN. Given a CAN $A = \langle \Pi, \eta, S\langle X\rangle, \Theta\rangle$, $Q \in \Pi$, sequence Y of Boolean constants such that $|Y| = \eta(Q)$, and an interpretation $\mathcal{I}$ for A, a *derivation tree* $T = \langle V, E, r, \mu\rangle$ for $Q\langle\mathcal{I}(Y)\rangle$ is a nonempty finite labeled tree such that V is the set of nodes in T, E is the set of arcs in T, r is a node in T called the *root*, and μ is a partial mapping from V to $\Theta^{\forall}$ such that

- for each internal node $v \in V$, if v has left and right children u_1 and u_2 respectively in T, then $\mu(v)$ is in the form of $P\langle X\rangle \rightarrow \max(P_1\langle X_1\rangle, P_2\langle X_2\rangle);$, $P\langle X\rangle \rightarrow \min(P_1\langle X_1\rangle, P_2\langle X_2\rangle);$, or $P\langle X\rangle \rightarrow P_1\langle X_1\rangle P_2\langle X_2\rangle;$, for some P, P_1, P_2 such that $\text{lhs}(\mu(u_1)) = P_1\langle X_1\rangle$ and $\text{lhs}(\mu(u_2)) = P_2\langle X_2\rangle$ respectively;
- for each leaf $v \in V$, either $\mu(v) = P\langle X\rangle \rightarrow (c)$; for some $P \in \Pi, c \in \mathcal{N}$; or $\mu(v)$ is undefined in Θ.
- $\text{lhs}(\mu(r)) = Q\langle\mathcal{I}(Y)\rangle$

A derivation tree $T = \langle V, E, r, \mu\rangle$ is *executable* iff one of the following three conditions is true.

- $\mu(r)$ is a rule like $P\langle X\rangle \rightarrow (c);$.

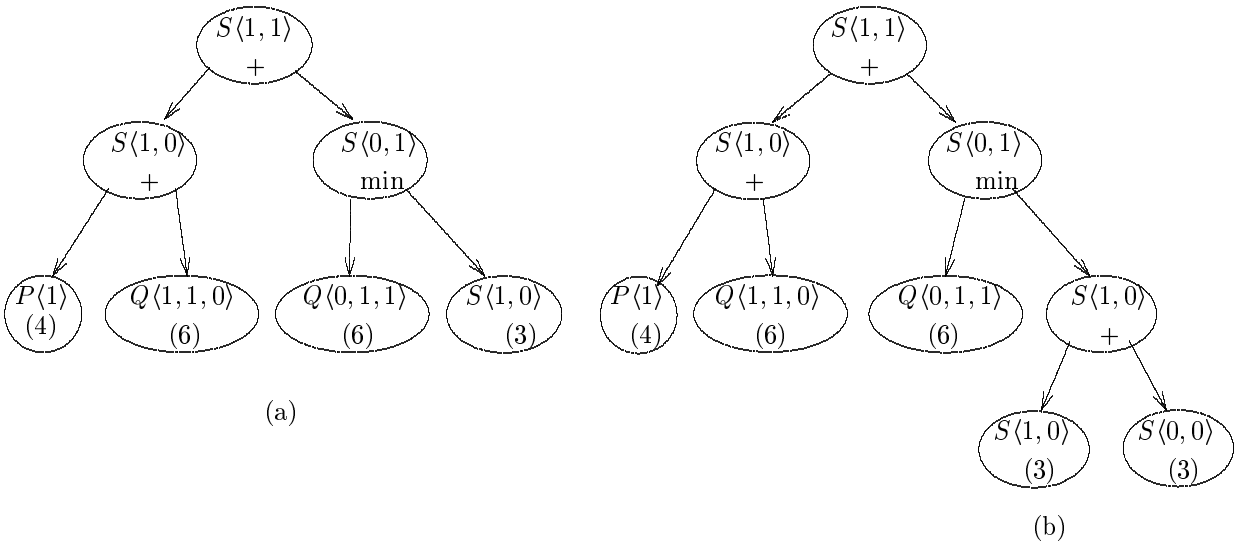

Figure 1: Two execution trees

- $\mu(r)$ is a rule like $P\langle X\rangle \rightarrow P_1\langle X_1\rangle P_2\langle X_2\rangle$; or $P \rightarrow \max(P_1\langle X_1\rangle, P_2\langle X_2\rangle)$; and both its left and right subtrees are executable.
- $\mu(r)$ is a rule like $P \rightarrow \min(P_1\langle X_1\rangle, P_2\langle X_2\rangle)$; and at least one of its two subtrees are executable.

An executable derivation tree is called an *execution tree*. We define the semantics of CAN on execution trees instead on derivation trees because intuitively given a min-rule, only one child has to have a derivation to generate a derivation from the rule. In an execution starting with rules like $P\langle X\rangle \rightarrow \min(P_1\langle X_1\rangle, P_2\langle X_2\rangle)$, in case $P_1\langle X_1\rangle$ does not have an execution, we request $[\![P_2]\!]_A \subseteq [\![P]\!]_A$. The intuition is that when a child procedure has no response at all, then the parent constructed with a min-rule has to rely fully on the other child for the successful completion of the task.

Example 2 *:* For the CAN given in example 1, we may have the two execution trees in Figure 1. In each node, we label a process type and the rule operator (blank for undefined rules, parenthesized numbers for constants, + for sequential, min for minimum, max for maximum). The labels of a node, its rule operator, and its children together record the construction of the relevant process. ‖

Each execution tree actually represents a computation of the root process symbol. Given an execution tree $T = \langle V, E, r, \mu\rangle$ with root r, we define its *execution time* $[\![T]\!]$ recursively on the structure of T.

- If $\mu(r)$ is undefined, then $[\![T]\!] = \infty$.
- If $\mu(r) = P\langle X\rangle \rightarrow (c)$ for some $c \in \mathcal{N}$, then $[\![T]\!] = c$.
- If r has two subtrees T_1, T_2, then either of the following three conditions is true : $\boxed{1}$ $\mu(r)$ is a

sequential rule and $[\![T]\!] = [\![T_1]\!] + [\![T_2]\!]$; $\boxed{2}$ $\mu(r)$ is a min-rule and $[\![T]\!] = \min([\![T_1]\!], [\![T_2]\!])$; $\boxed{3}$ $\mu(r)$ is a max-rule and $[\![T]\!] = \max([\![T_1]\!], [\![T_2]\!])$.

For convenience, from now on, internal nodes labeled with addition (sequential) rules, minimum rules, and maximum rules are respectively called *addition nodes*, min-*nodes* and max-*nodes*.

Example 3 *:* It is obvious that each execution tree has an execution time. For the two trees shown in Figure 2, execution times are 13 and 16 respectively. ||

Now the semantics of CAN is defined to be the set of execution times of the starting process invocation.

Definition 1 *: Semantics of CAN*

Suppose we are given a CAN $A = \langle \Pi, \eta, S\langle X\rangle, \Theta\rangle$. For each $P \in \Pi$ and Boolean sequence Y such that $|Y| = \eta(P)$, we let

$$[\![P\langle Y\rangle]\!]_A = \{[\![T]\!] \mid T \text{ is an execution tree for } P\langle Y\rangle \text{ in } A; [\![T]\!] \neq \infty\}$$

Process $P\langle Y\rangle$ is said to have an execution if $[\![P\langle Y\rangle]\!]_A \neq \emptyset$. The semantics of A, denoted as $[\![A]\!]$, is defined to be $[\![S\langle X\rangle]\!]_A$. ||

We here use several examples to show the benefit we may get by applying CAN theory properly. Subsection 3.3 shows how to predict the execution time pattern of a recursive tree-traversing procedure when no knowledge of the tree size is assumed.

Subsection 3.4 discusses how to schedule the real-time resource requests of two processes constructed from recursive procedures. In classic verification theory based on state sequences, such a verification problem is not identified as decidable because the intersection emptiness of two context-free languages is undecidable[12]. CAN has the advantage here since its execution time set will be shown to be semilinear in section 4 and semilinear set intersection is computable.

Subsection 3.5 shows CAN's capability in verifying systems with dynamic process instantiation and destruction. In traditional verification theory, when users are allowed such a capability, the verification problem can easily become undecidable since instantiation and destruction of processes can respectively simulate the increment and decrement operations of two-counter machines. Example 6 shows that as long as users obey the CAN paradigm, timing predictability is still feasible.

3.3 Analysis of recursive tree search

Example 4 *:* Suppose we want to traverse a binary tree stored in a file system with the procedure in Table 1. Each node in the tree can be painted green or blue. Procedure Traverse() processes each node

Traverse(T)
/* The left and right children of T are T.left and T.right respectively.
* T may be either green or blue.
*/ {
(1) If T is null, return.
(2) If T is green and internal, then paint T.left green and process for 5 time units.
(3) If T is blue and internal, then paint T.right blue and process for 25 time units.
(4) Traverse(T.left)
(5) Traverse(T.right)
}

Table 1: A recursive tree traversing procedure

according to their color and position in the tree. The color of each node also affect its children's color. This emulates some interaction between two generations.

But we can emulate the procedure by the following CAN. We use Boolean constant 0 to represent green while 1 to represent blue.

$$T\langle x\rangle \to (0); \quad T\langle 0\rangle \to (5)T\langle 0\rangle T\langle x\rangle; \quad T\langle 1\rangle \to (25)T\langle x\rangle T\langle 1\rangle; \quad T\langle 1\rangle \to (25)T\langle 1\rangle;$$

The first rule is straightforward. The second rule says a green node is processed with 5 time units and paints its left child. The third rule says that for a blue node, it needs 25 time units of processing and if its right child is internal, then the child needs some execution time as a blue node. The fourth rule says that for a blue node, it needs 25 time units of processing and if its right child is external, then the child needs no execution time.

By using the classic result of R. Parikh[15], we can calculate its admissible execution times as a semilinear characteristic set. In this way, given any execution time c, the user can predict if there is any tree-configuration which needs c time units to traverse. This kind of analysis can thus be used to schedule the resource utilization in real-time systems. ||

3.4 Analysis of schedulability

The previous example falls entirely in the extent of R. Parikh's work[15], thus BCAN, and does not use the parallel operations of CAN. The following example gives us a taste about how the parallel operators can be useful.

Example 5 : Suppose we have two stringent processes A and B which keep on posting sporadic scheduling requests. They are stringent in the sense that on receiving their scheduling request, the scheduler does not have too much time slack to juggle with. The only thing we know is that for each process, in between its two successive scheduling requests, it executes some internal recursive operation described by CAN's which is of no interest to the outside observers.

Suppose that we are lucky that the descriptions for A and B both fall in class CAN (or PCAN, SCAN in section 5) and their semilinear representations are $3+5i$ and $7+25j$ respectively. Due to the stringent timing requirements of the scheduling, one safe solution is to reserve resources for the two processes according to their execution time pattern. Since for all $a, b \in \mathcal{N}$, the intersection of $\{a+3+5i \mid i \in \mathcal{N}\}$ and $\{b+7+25j \mid j \in \mathcal{N}\}$ can be calculated efficiently, our analysis guarantees mutual exclusion safety of such a scheduling policy. ||

We would like to point out that although the last example seems to put too much restrictions on the application of CAN, in fact in complex software systems, there is much room for timing constants adjustment. Thus it is possible to use CAN theory to guide the constant adjustment in order to maintain mutual exclusion.

3.5 Analysis of dynamic processes

The following example shows how to describe dynamically instantiated processes in CAN. Traditional algorithmic analysis of real-time systems usually assumes a static scenario of process lives, so no processes are instantiated or destroyed in the middle of computation.

Example 6 : One parallel programming style advocates the use of **COBEGIN** and **COEND** (or FORK and JOIN). For example, a user may write code like

$$\textbf{COBEGIN } P\langle\text{commit}\rangle; Q\langle\text{commit}\rangle \textbf{ COEND}$$

or

$$\textbf{COBEGIN } P\langle\text{abort}\rangle; Q\langle\text{abort}\rangle \textbf{ COEND}$$

to synchronize behaviors of different concurrent processes. The codes are executed by simultaneously executing program P and Q and completed when both P and Q are finished. When there is no interaction except the parameter-sharing between processes P and Q, the code falls in CAN subclass and its execution time pattern is semilinear even with the presence of software recursion. ||

4 Deriving semilinear descriptions of CAN

The algorithm to generate the semilinear description is presented and proved in four steps. First, we show that CAN is equivalent to CAN without parameters. Second, we establish that CAN is equivalent to CAN without max-rules. Then we present an algorithm to tell if the execution time set of a CAN is of finite size. Finally we constructively prove that all CAN with no parameters and no max-rules are semilinear by further reducing them to BCAN. The algorithm for calculating the semilinear expressions of CAN is derived based on the constructive proofs.

4.1 CAN with no parameters

For a fixed number of Boolean parameter positions, we can compute all the Boolean parameter value combinations. Given an $n \in \mathcal{N}$, we let $\{0,1\}^n$ be the set of all n-bit Boolean vectors. For example, $\{0,1\}^2 = \{\langle 0,0\rangle, \langle 0,1\rangle, \langle 1,1\rangle, \langle 1,0\rangle\}$.

Given a CAN $A = \langle \Pi, \eta, S\langle X\rangle, \Theta\rangle$, we generate $\check{A} = \langle \check{\Pi}, \check{\eta}, S\langle X\rangle, \check{\Theta}\rangle$ such that $\check{\Pi} = \{S\} \cup \{P^\chi \mid P \in \Pi; \chi \in \{0,1\}^{\eta(P)}\}$, $\check{\eta}(S) = \eta(S) \wedge \forall P^\chi \in \check{\Pi}(\check{\eta}(P^\chi) = \eta(P))$, and $\check{\Theta} = \Theta^\forall \cup \{S \to S^\chi \mid \chi \in \{0,1\}^{\eta(S)}\}$. The followig lemma shows that $[\![A]\!] = [\![\check{A}]\!]$.

LEMMA 1 *: Given CAN $A = \langle \Pi, \eta, S\langle X\rangle, \Theta\rangle$, for every execution tree T for A, there is an isomorphic execution tree $\check{T}$ for $\check{A}$ such that for every node v in T and its corresponding node $\check{v}$ in $\check{T}$, if $lhs(\mu(v)) = P\langle Y\rangle$, then $lhs(\mu(\check{v})) = P^Y$.*

Proof : Straightforward from the construction of $\check{A}$. ‖

4.2 CAN with no max-rules

From now on, we shall assume that all CAN's given to us have no parameters. Our presentation plan can be broken down to two steps. First we shall use lemma 2 to show that the minimum execution time of a process type can be generated by a tree shorter than $3|\Pi| - 1$. This gives us an algorithm to determine the minimum weights of process types. Then we shall present a reduction from arbitrary CAN to CAN without max-rules and prove its correctness.

The proofs demand the definition of substructures in execution trees. Given an execution tree T, the *contributing tree* C of T is the tree structure in T that really contributes to the weight of T. C can be constructed from T using the procedure Contribute() in table 2. It is easy to see that along any path in the contributing tree, the weight of a subtree cannot be smaller than that of its child subtrees in the contributing tree.

Contribute(T) /* The root, left, and right subtrees are r, T_1, T_2 respectively. */ {
(1) If r is a leaf, return T.
(2) Else if r is labeled with an addition rule,
return tree$(r, \text{Contribute}(T_1), \text{Contribute}(T_2))$.
(3) Else if r is labeled with a minimum or a maximum rule and $[\![T]\!] = [\![T_1]\!] = [\![T_2]\!]$,
return tree$(r, \text{Contribute}(T_1), \text{Contribute}(T_2))$.
(4) Else if r is labeled with a minimum or a maximum rule and $[\![T]\!] = [\![T_1]\!] \neq [\![T_2]\!]$,
return tree$(r, \text{Contribute}(T_1))$.
(5) Else if r is labeled with a minimum or a maximum rule and $[\![T]\!] = [\![T_2]\!] \neq [\![T_1]\!]$,
return tree$(r, \text{Contribute}(T_2))$.
}

• tree$(r, T_1, \ldots, T_n)$ returns a new tree whose root is r and whose subtrees from left to right are $T_1, \ldots, T_n$ respectively.

Table 2: Procedure for generating the contributing tree

Prune$_1(T)$ {
(1) Identify the contributing tree C in T.
(2) Identify two nodes, v_1 and v_2, along a path in C from the root to a leaf such that
- v_1 is the ancestor of v_2;
- v_1 and v_2 are labeled with rules with the same left-hand-side.

(3) Replace the subtree rooted at v_1 by the one rooted at v_2.
(4) Return the modified T.
}

Table 3: One procedure for pruning a tree

LEMMA 2 : *Given a CAN* $A = \langle \Pi, \eta, S, \Theta \rangle$ *with no parameters, for each* $P \in \Pi$, *if there is an execution tree for* P, *then there is an execution tree of height* $< 3|\Pi| - 1$ *and weight* $\min([\![P]\!]_A)$ *for* P.

Proof : Now assume that T is an execution tree with the minimum weight of all execution trees for P. We repeat procedure Prune$_1()$ described in table 3 until the contributing tree is shorter than $|\Pi|$. It is clear that after this pruning process, the weight of T cannot increase. For all other subtrees of T not in C after the final pruning iteration, we can again perform another pruning procedure described in table 4 on them until all of them are shorter than $2|\Pi|$. Note that in Prune$_2()$, we have to be careful to maintain the executability of pruned trees. This forced the search of path shorter than $2|\Pi|$. This finally transforms T into an execution tree for P with minimum weight and height $< 3|\Pi| - 1$. ‖

Given a CAN $A = \langle \Pi, \eta, S, \Theta \rangle$, we shall derive $\bar{A} = \langle \bar{\Pi}, \bar{\eta}, S, \bar{\Theta} \rangle$ such that there is no max-rule in $\bar{\Theta}$

Prune$_2(T)$ {
(1) Identify two nodes, v_1 and v_2, along a path in T from the root to a leaf such that
• v_1 is the ancestor of v_2;
• v_1 and v_2 are labeled with rules with the same left-hand-side.
• v_1 roots an executable tree iff v_2 also roots an executable tree.
(2) Replace the subtree rooted at v_1 by the one rooted at v_2.
(3) Return the modified T.
}

Table 4: Another procedure for pruning a tree

and $[\![\bar{A}]\!] = [\![A]\!]$. Lemma 2 shows that given a process type P (Note we have assumed that A has no parameters), we can find the minimum weight of P's execution trees by enumerating P's executing tree shorter than $3|\Pi| - 1$. Given $A = \langle \Pi, \eta, S, \Theta \rangle$ and $P \in \Pi$, we let $\min_A(P)$ be the minimum execution time of all execution trees for $P \in \Pi$. When P does not have an execution tree, we let $\min_A(P) = \infty$. We then let $M_A = \max\{\min_A(P) \mid P \in \Pi\}$. The new CAN is defined to be $\bar{A} = \langle \bar{\Pi}, \bar{\eta}, S, \bar{\Theta} \rangle$ with

$$\bar{\Pi} \stackrel{\text{def}}{=} \Pi \cup \{P_i \mid P \in \Pi; i \in \mathcal{N}; 0 \le i < M_A; \} \cup \{P_{\ll} \mid P \in \Pi\},$$

where $P_{\ll}$ is a newly introduced nonterminal symbol, $\bar{\eta}$ maps everything to zero, and $\bar{\Theta} \stackrel{\text{def}}{=}$

$$\begin{pmatrix} \{P \to P_i \mid P \in \Pi; i \in \mathcal{N}; 0 \le i < M_A; \} & \cup \{P_i \to \min(Q_i, R_k) \mid P \to \min(Q, R) \in \Theta; i \le k\} \\ \cup \{P \to P_{\ll} \mid P \in \Pi\} & \cup \{P_i \to \min(Q_j, R_i) \mid P \to \min(Q, R) \in \Theta; i \le j\} \\ \cup \{P_c \to (c) \mid P \to (c) \in \Theta; c < M_A\} & \cup \{P_i \to \min(Q_{\ll}, R_i) \mid P \to \min(Q, R) \in \Theta; 0 \le i < M_A\} \\ \cup \{P_{\ll} \to (c) \mid P \to (c) \in \Theta; c \ge M_A\} & \cup \{P_{\ll} \to \min(Q_{\ll}, R_{\ll}) \mid P \to \min(Q, R) \in \Theta\} \\ \cup \{P_i \to Q_j R_k \mid P \to QR \in \Theta; i = j + k\} & \cup \{P_i \to \min(Q_i, R_{\ll}) \mid P \to \min(Q, R) \in \Theta; 0 \le i < M_A\} \\ \cup \{P_{\ll} \to Q_j R_k \mid P \to QR \in \Theta; j + k \ge M_A\} & \cup \{P_{\ll} \to Q_{\ll} \mid P \to \max(Q, R) \in \Theta; [\![R]\!]_A \ne \emptyset\} \\ \cup \{P_{\ll} \to Q_{\ll} R_k \mid P \to QR \in \Theta; 0 \le k < M_A\} & \cup \{P_{\ll} \to R_{\ll} \mid P \to \max(Q, R) \in \Theta; [\![Q]\!]_A \ne \emptyset\} \\ \cup \{P_{\ll} \to Q_j R_{\ll} \mid P \to QR \in \Theta; 0 \le j < M_A\} & \cup \{P_i \to Q_i \mid P \to \max(Q, R) \in \Theta; \min_A(R) \le i < M_A\} \\ \cup \{P_{\ll} \to Q_{\ll} R_{\ll} \mid P \to QR \in \Theta\} & \cup \{P_i \to R_i \mid P \to \max(Q, R) \in \Theta; \min_A(Q) \le i < M_A\} \end{pmatrix}$$

LEMMA 3 *: Any CAN A with no parameters has the same execution time set as that of the $\bar{A}$ defined in the above.*

Proof : ($\Longrightarrow$) We want to show by induction that for every $P \in \Pi$ and $w \in \mathcal{N}$, if there is an execution tree for P of execution time w, then there is such an execution tree in $\bar{A}$. In the basic case, we have an execution like $P \to (c)$ in A. If $c \ge M_A$, then we have $P \to P_{\ll}$ and $P_{\ll} \to (c)$. Otherwise, we have $P \to P_c$ and $P_c \to (c)$. Now assume that this direction of the lemma is true for all execution tree in A of height $\le h$. Given an execution tree in A of height $h + 1$, there are two cases to consider. First suppose the root is constructed from a sequential rule $P \to QR$ and the two subtrees have execution times c and

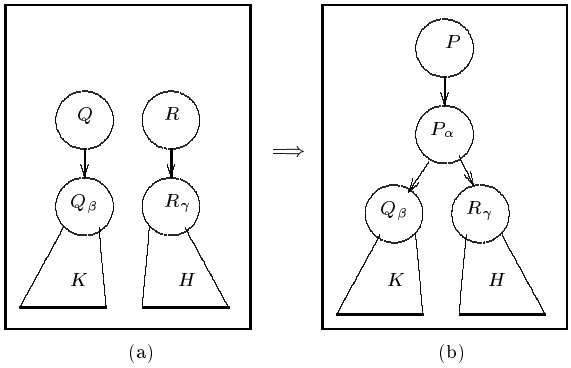

Figure 2: Tree constructions for $\bar{A}$

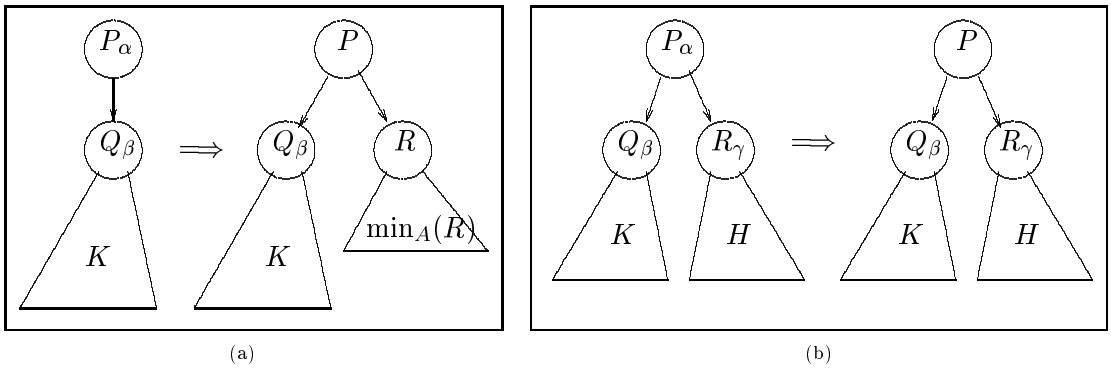

Figure 3: Two structure transformations

d respectively. According to the inductive hypothesis, we know that there are the two execution trees in $\bar{A}$ shown in Figure 2(a) for some β and γ. From these two trees, we can thus construct the tree in Figure 2(b) for some α and prove the case. The case when the root corresponds to a max- or a min-rule can also be proven by similar construction.

($\Longleftarrow$) Given an execution tree T in $\bar{A}$ for a $P \in \Pi$, we can construct an execution tree in A for P with the same execution time. Note in $\bar{A}$, process types without subscripts, like P, only appear in the left-hand-sides of the rules. The construction goes in the following two steps. First we remove the root node from T. Second, we relabel the internal nodes with rules in Θ according to the two transformations depicted in figure 3. Figure 3(a) is for max-rules like $P \rightarrow \max(Q, R)$ while figure 3(b) is for rules like either $P \rightarrow QR$ or $P \rightarrow \min(Q, R)$. Note in figure 3(a), we use the drawing of a triangle headed by circled R with $\min_A(R)$ inside as the execution tree of R with the minimum execution time for R. We do the transformation for each original node (labeled with process types with subscripts) in the original

execution tree T. The result after the transformation is an execution tree for A and can be shown by induction on the structure to be of the same execution time as T. This finishes the proof. ||

4.3 Finiteness of the sizes of CAN execution time sets

According to lemma 3, from now on, we shall assume without loss of generality that all CAN's given to us have no parameters and no max-rules. Here we shall develop the necessary and sufficient condition for a process to have infinitely many different execution times. We have the following plan of presentation in this subsection. First, we use definitions 2 and 3 to construct the necessary and sufficient condition which is called *unboundedness condition*. Then we use lemma 4 to prove that the condition is indeed necessary and sufficient. Finally, we use the constructive proof in lemma 4 to develop a PSPACE algorithm to detect the unboundedness condition.

We define the following structures in an execution tree.

Definition 2 : **Bone trees** Given an execution tree $T = \langle V, E, r, \mu \rangle$ for $P \in \Pi$ in $A = \langle \Pi, \eta, S, \Theta \rangle$, a *bone tree* $B = \langle \bar{V}, \bar{E}, r, \mu \rangle$ of T is a substructure in T satisfying the following conditions.

- $\bar{V} \subseteq V$
- $\bar{E} = \{(N, N') \mid (N, N') \in E; N, N' \in \bar{V}\}$
- If $N \in \bar{V}$ and $\mu(N)$ is a rule like $P \rightarrow QR$, then one of N's children is in $\bar{V}$.
- If $N \in \bar{V}$ and $\mu(N)$ is a rule like $P \rightarrow \min(Q, R)$, then N's two children are both in $\bar{V}$. ||

Definition 3 : **Unboundedness condition** Given $P \in \Pi$ in $A = \langle \Pi, \eta, S, \Theta \rangle$, we say P satisfies the *unboundedness condition* iff there is an execution tree T of P and a bone tree B in T such that along all paths from root to leaves in B, there is a path segment $v_i, v_{i+1}, \ldots, v_j$, with $i < j$, such that

- v_i and v_j are labeled with rules with the same left-hand-side; and
- there is an $i \leq k < j$ such that v_k is labeled with a sequential concatenation rule and the child of v_k not in the path roots an execution tree of nonzero execution time. ||

We first want to prove lemma 4.

LEMMA 4 : *Given a CAN* $A = \langle \Pi, \eta, S, \Theta \rangle$*, with no* max*-rules and no parameters, and* $P \in \Pi$*,* $|[\![P]\!]_A| = \infty$ *iff* P *satisfies the unboundedness condition.*

Proof : ($\Longrightarrow$) Let ω be the biggest constant used in the definition of A. Since $|[\![P]\!]_A|$ is infinite, there must be an execution tree T for P heavier than $2^{|\Pi|}\omega$. Now we shall constructively identify a bone tree B in T which satisfies the unboundedness condition. The construction is accomplished by procedure Bone() described in table 5. For example, both of the execution trees in figure 1 generate the same

Bone(T) /* The root, left, and right child subtrees of T are r, T_1, and T_2. */ {
(1) If r is a leaf, return T.
(2) If r is an addition node, then
If $[\![T_1]\!] \geq [\![T_2]\!]$, return tree$(r, \text{Bone}(T_1))$; else return tree$(r, \text{Bone}(T_2))$;
(3) If r is a min-node, return tree$(r, \text{Bone}(T_1), \text{Bone}(T_2))$;
}

- tree$(r, T_1, \ldots, T_n)$ returns a new tree whose root is r and whose subtrees from left to right are $T_1, \ldots, T_n$ respectively.

Table 5: Procedure for constructing a bone tree

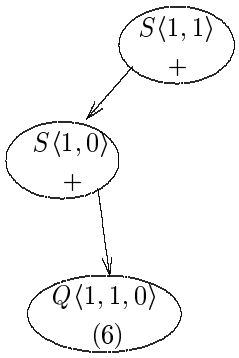

Figure 4: A bone tree

bone tree depicted in figure 4 through procedure Bone(). Now we want to show that B indeed satisfies the unboundedness condition.

We call a node *heavy* if it is labeled with an addition rule and both of its children root subtrees of nonzero weights. Other nodes are called *light*. We want to prove that along each path in B, there are at least $|\Pi| + 1$ heavy nodes. Note that the weight of a subtree T' is broken down into two nonzero weights of the corresponding child subtrees only when the root of T' is a heavy node. Also at each heavy node in the execution of procedure Bone(), we choose to include only the heavier child subtree. Thus it is clear that after passing each heavy node, the weight of the subtree is reduced to a value no less than half of the weight of its parent tree. Also if we branch through a min-node, the weight of the child subtrees can not be smaller than the weight of its parent tree. Since $[\![T]\!] > 2^{|\Pi|}\omega$, it is clear that along any path of B, there will be at least $|\Pi| + 1$ heavy nodes.

Finally, of the $|\Pi| + 1$ heavy nodes along each path in B, there are two labeled with rules with the same left-hand-side. Thus this direction is proven.

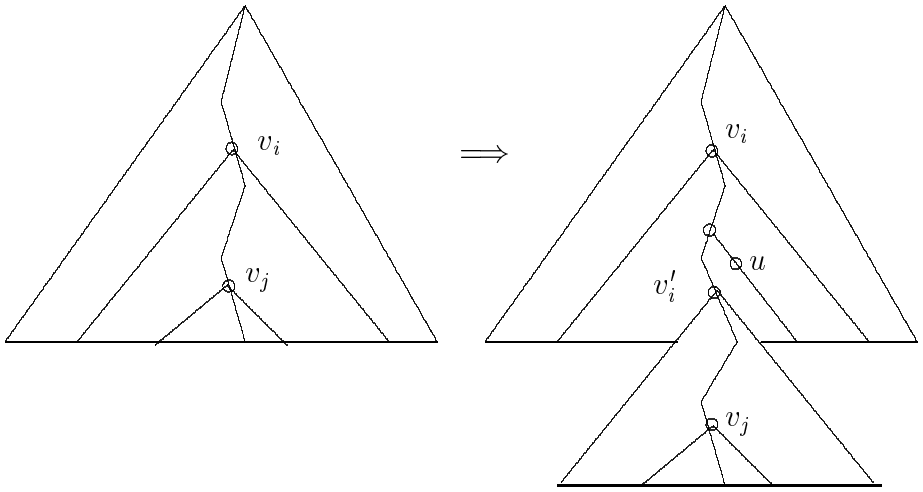

Figure 5: Extending an execution tree

($\Longleftarrow$) If there is such an execution tree T, we shall show that from T we can construct another execution tree T' such that $[\![T']\!] > [\![T]\!]$ and T' also satisfies the unboundedness condition. We identify in B a subgraph C_B which contributes to the weight of T. That is for a node N in B labeled with a min-rule, we only keep in C_B the child subtrees of N which have the same weights as the one rooted at N. We call such a subgraph a *contributing bone tree*.

Along every root-leaf path in C_B, we identify nodes v_i and v_j as described in the unboundedness condition. Then we replace the subtree rooted at v_j by the one rooted at v_i. The replacement for one pair is shown in figure 5. We perform such a replacement for each root-leaf path in C_B. Note in C_B, the weight of the tree rooted at v_i is greater than that of the tree rooted at v_j. After the replacement, the weight of the tree rooted at N_i' is lifted to the weight of the tree originally rooted at v_i. Since the path contributes the weight of the whole tree, it can be shown that the result new tree T' is heavier than $[\![T]\!]$ and satisfies the unboundedness condition. ||

To utilize lemma 4, one question still need to be answered: how do we determine if a nonterminal has a nonzero execution time? We can assign a variable ν_P to each $P \in \Pi$ which may have one of three values : 0 (has a zero weight), 1 (has a nonzero weight), and $-\infty$ (has no execution tree). Then we feed these variables through the rules in Θ iteration by iteration. After iteration k, we know if a particular nonterminal has a nonzero weight execution tree of height k. Within at most $2|\Pi|$ iterations, a fixpoint will be reached and we can tell which nonterminals have nonzero weights. This also indirectly shows that the nonzero execution weight tree found is of height $2|\Pi|$ at most.

```
Unbounded(A, P) /* A = ⟨Π, η, S, Θ⟩ */
{
(1) Let Γ = {[P, |Π|]};
(2) Repeat {
  (1) Nondeterministically choose an element [Q, c] from Γ;
  (2) Nondeterministically choose θ ∈ Θ such that lhs(θ) = Q;
  (3) If there is no such θ, return FALSE; else {
    (1) Delete [Q, c] from Γ;
    (2) If θ is Q → R1R2;, nondeterministically choose one of the following to perform. {
      (1) If ν_R1 = 0, then add [R2, c] to Γ;
        else if ν_R1 = −∞, then return FALSE;
        else if ν_R1 = 1 and c > 0, then add [R2, c − 1] to Γ;
        else if ν_R1 = 1 and c = 0 and ν_R2 = −∞, then return FALSE;
      (2) If ν_R2 = 0, then add [R1, c] to Γ;
        else if ν_R2 = −∞, then return FALSE;
        else if ν_R2 = 1 and c > 0, then add [R1, c − 1] to Γ;
        else if ν_R2 = 1 and c = 0 and ν_R1 = −∞, then return FALSE;
      }
    (3) If θ is Q → min(R1, R2), then add [R1, c], [R2, c] to Γ;
    (4) If Γ = ∅, return TRUE;
    (5) If Γ has been visited before, return FALSE;
    }
  }
}
```

Table 6: Procedure for determining the unboundedness condition

Now we propose to use the first part of the constructive proof for lemma 4 to develop an algorithm to tell if a process satisfies the unboundedness condition. The algorithm is embodied in procedure Unbounded() in table 6. Given the target process P in $A = \langle \Pi, \eta, S, \Theta \rangle$, Unbounded() nondeterministically searchs for the empty set node. In the repetition loop, counter c decrements only when we have passed through a heavy node. Also, no new element is added to Γ if $c = 0$. This happens when we have traversed through $|\Pi| + 1$ heavy nodes along a path. Thus when we reach the empty set node, we know that along all paths, we have traversed through $|\Pi| + 1$ heavy nodes and P satisfies the unboundedness condition.

A nondeterministic implementation of Unbounded() needs only polynomial space. But the terminating condition at statement (2.3.5) has to be changed to search length test of $2^{|\Pi|(|\Pi|+1)}$ since there can only be $2^{|\Pi|(|\Pi|+1)}$ distinct configurations of Γ.

Bound(A) /* $A = \langle \Pi, \eta, S, \Theta \rangle$ */
{
(1) For each $P \in \Pi$, if Big(P), then $\Phi_P = \infty$, else $\Phi_P = -\infty$
(2) Loop for $2^{|\Pi|}\omega|\Pi| + 1$ times. {
  (1) For each rule $P \to (c) \in \Theta$, if $c > \Phi_P$, let $\Phi_P := c$;
  (2) For each rule $P \to QR \in \Theta$, if $\Phi_Q + \Phi_R > \Phi_P$, let $\Phi_P := \Phi_Q + \Phi_R$;
  (3) For each rule $P \to \min(Q, R) \in \Theta$, if $\min(\Phi_Q, \Phi_R) > \Phi_P$, let $\Phi_P := \min(\Phi_Q, \Phi_R)$;
 }
}

Table 7: Procedure for computing the maximum execution times

4.4 Semilinear expressions of CAN

For convenience and conciseness of presentation, we let $\text{Big}(P) \stackrel{\text{def}}{=} (|[\![P]\!]_A| = \infty \vee |[\![P]\!]_A| = 0)$ and $\text{Small}(P) \stackrel{\text{def}}{=} (0 < |[\![P]\!]_A| \neq \infty)$.

With the algorithm in the last subsection, we can replace rules like $P \to \min(Q, R)$ by $P \to Q, P \to R$ if $\text{Big}(Q) \wedge \text{Big}(R)$ is true. Now the remaining issues is how we are going to handle the min-rule when either $|[\![Q]\!]|$ or $|[\![R]\!]|$ is a nonzero integer. Our strategy is to first compute the maximum execution time of all P ($\max [\![P]\!]$) with $\text{Small}(P)$. Then we use this as upper bound to help us enumerate all the rule compositions, as we have done in the reduction in subsection 4.2.

The maximum execution times of all process types can be calculated with Procedure Bound(A) in table 7. It is known from lemma 4 that for a Q with $\text{Small}(Q)$, $\max [\![Q]\!] \leq 2^{|\Pi|}\omega$. Thus wihtin $2^{|\Pi|}\omega|\Pi|+1$ iterations, a fixed point will be reached and the maximum execution times of all process types can be found. At the end of execution of procedure Bound(A), for each $P \in \Pi$ with $\text{Small}(P)$, $\max [\![P]\!] = \Phi_P$.

Now we define yet another reduction to reduce a CAN with no parameters and no max-rules into a BCAN $\hat{A} = \langle \hat{\Pi}, \eta, S, \hat{\Theta} \rangle$ with $\hat{\Pi} = \{S\} \cup \left\{ P_c \mid 0 \leq c \leq 2^{|\Pi|}\omega \right\} \cup \{P_{\ll} \mid P \in \Pi\}$ and

$$\hat{\Theta} \stackrel{\text{def}}{=} \left(\begin{array}{ll} & \{S \to S_i \mid 0 \le i \le 2^{|\Pi|}\omega\} \\ \cup & \{S \to S_{\ll}\} \\ \cup & \{P_c \to (c) \mid P \to (c) \in \Theta\} \\ \cup & \{P_i \to Q_j R_k \mid P \to QR \in \Theta; i = j + k \le 2^{|\Pi|}\omega\} \\ \cup & \{P_{\ll} \to Q_j R_k \mid P \to QR \in \Theta; j + k > 2^{|\Pi|}\omega\} \\ \cup & \{P_{\ll} \to Q_{\ll} R_k \mid P \to QR \in \Theta; 0 \le k \le 2^{|\Pi|}\omega\} \\ \cup & \{P_{\ll} \to Q_j R_{\ll} \mid P \to QR \in \Theta; 0 \le j \le 2^{|\Pi|}\omega\} \\ \cup & \{P_{\ll} \to Q_{\ll} R_{\ll} \mid P \to QR \in \Theta\} \\ \cup & \{P_i \to Q_i \mid P \to \min(Q, R) \in \Theta; \text{Small}(R); i \le \max[\![R]\!]\} \\ \cup & \{P_i \to R_i \mid P \to \min(Q, R) \in \Theta; \text{Small}(Q); i \le \max[\![Q]\!]\} \\ \cup & \{P_i \to Q_i \mid P \to \min(Q, R) \in \Theta; \text{Big}(R); 0 \le i \le 2^{|\Pi|}\omega\} \\ \cup & \{P_i \to R_i \mid P \to \min(Q, R) \in \Theta; \text{Big}(Q); 0 \le i \le 2^{|\Pi|}\omega\} \\ \cup & \{P_{\ll} \to Q_{\ll} \mid P \to \min(Q, R) \in \Theta; \text{Big}(R)\} \\ \cup & \{P_{\ll} \to R_{\ll} \mid P \to \min(Q, R) \in \Theta; \text{Big}(Q)\} \end{array} \right)$$

THEOREM 5 *: Suppose we are given a CAN $A = \langle \Pi, \eta, S, \Theta \rangle$ without no parameters and max-rules and a BCAN $\hat{A}$ constructed according to the just-mentioned reduction. Then $[\![\hat{A}]\!] = [\![A]\!]$.*

Proof : True according to the results in the last two subsections. ‖

Since all our reductions and proofs are constructive, an intuitive algorithm to derive the semilinear expressions of any given CAN is in the following four steps : (1) Transform the CAN to a CAN without parameters. (2) Transform the CAN without parameters to a CAN without max-rules. (3) Transform the CAN without max-rules to a BCAN in Parikh's subclass. (4) Use Parikh's result [15] to derive the semilinear expressions of the BCAN.

5 Other possibilities for CAN

The parallel composition of CAN adopts the minimum and maximum semantics. One other possibility is in the flavor of rendezvous. That is we can design a parallel rule $P\langle X \rangle \to P_1\langle X_1 \rangle \| P_2\langle X_2 \rangle$; such that process $P\langle X \rangle$ has an execution tree if $P_1\langle X_1 \rangle$ and $P_2\langle X_2 \rangle$ has executions of the same execution time. Intuitively, this means $P\langle X \rangle$ can be invoked by simultaneously invoking $P_1\langle X_1 \rangle$ and $P_2\langle X_2 \rangle$ and $P\langle X \rangle$ is fulfilled when $P_1\langle X_1 \rangle$ and $P_2\langle X_2 \rangle$ are fulfilled at the same time.

Along this semantics, we have designed two algebra, *PCAN (Parallel CAN)* and *SCAN (Stratified CAN)*. PCAN corresponds to the *Basic Parallel Process (BPP)* of process algebra in that the sequential concatenation rules is forbidden except for the case of $P \to (c_1)P_1(c_2)$; for some integer constants c_1 and c_2.

The second subclass, *Stratified CAN (SCAN)*, merges advantage of both Parikh's classic result (BCAN) and that of PCAN to allow for the finite number of alternations between rendezvous and

sequential concatenation.

We shall give constructive proof for the semilinearity of PCAN and SCAN. The algorithm for computing the semilinear expressions of any PCAN or SCAN systems will also be described.

Finally we try to connect to the theory of recurrence theory through the device of integer parameter sharing. The result is DRCAN which can be used to analyze the execution of complex loops. However the execution time sets of DRCAN are not all semilinear.

With the technique presented in subsection 4.1, we can translate each of PCAN, SCAN, and DRCAN into an equivalent algebra without Boolean parameters. Thus in the following discussion, we shall assume that all algebrae given to us are without Boolean parameters.

5.1 Parallel CAN (PCAN)

Parallel CAN has the following syntax rules.

$$P \to (c); \qquad P \to (c_1)P_1(c_2); \qquad P \to P_1 \| P_2;$$

Due to the similarity shared with the semantic definition of CAN, we shall only briefly describe the semantics of PCAN. Given a rule like $P \to (c_1)P_1(c_2);$, process P is executed by doing some preprocessing of c_1 time units, then invoking P_1, and finally after the fulfillment of P_1, do some postprocessing of c_2 time units. Given a rule like $P \to P_1 \| P_2;$, process P is executed by invoking P_1 and P_2 simultaneously and fulfilled by the simultaneous fulfillment of P_1 and P_2 respectively.

The semantics of PCAN can also be defined with execution trees. Not all execution trees are legal because an internal node labeled with a parallel rule may have two subtrees of unequal execution times.

Due to the associativity and commutativity of additions, numerically $(c_1)P_1(c_2)$ is equivalently to tail procedure-call $(c_1 + c_2)P_1$. We use the following rewriting rules on PCAN to analyze its execution time set.

1. $(c_1)(c_2) \Longrightarrow (c_1 + c_2)$
2. Suppose we are given $c = \min\{c_1, \ldots, c_m\} > 0$ and for each $1 \leq i \leq m$, Q_i is either ϵ (empty process) or a process type name. $((c_1)Q_1) \| \ldots \| ((c_m)Q_m) \Longrightarrow (c)(((c_1 - c)Q_1) \| \ldots \| ((c_m - c)Q_m))$.
3. Given a rule $P \to (c)$, $(0)P \Longrightarrow (c)$.
4. Given a rule $P \to (c_1)P_1(c_2)$, $(0)P \Longrightarrow (c_1 + c_2)P_1$.
5. Given a rule $P \to P_1 \| P_2$, $(0)P \Longrightarrow ((0)P_1) \| ((0)P_2)$.

We let $\stackrel{*}{\Longrightarrow}$ be the transitive relation of $\Longrightarrow$. We let

$$[\![(d)(((c_1)Q_1) \| \ldots \| ((c_m)Q_m))]\!] = \{d' \mid (d)(((c_1)Q_1) \| \ldots \| ((c_m)Q_m)) \stackrel{*}{\Longrightarrow} (d')((0) \| \ldots \| (0))\}$$

The definition is general enough to cover that of execution time set of PCAN. Given a PCAN $A = \langle \Pi, \eta, S, \Theta \rangle$, if $(0)S \stackrel{*}{\Longrightarrow} (d)(((c_1)Q_1)\| \ldots \|((c_m)Q_m))$, then $((c_1)Q_1)\| \ldots \|((c_m)Q_m)$ represents a global snapshot of the process expansions of all concurrent threads. Let L_A be twice the biggest constant used in rules in Θ. According to the conditions of rewriting rules 2, 3, and 4, we find that c_i is bounded by L_A for all $1 \leq i \leq m$.

Moreover, the following lemma gives the finite characteristic of the analysis of PCAN.

LEMMA 6 *: Given a PCAN $A = \langle \Pi, \eta, S, \Theta \rangle$, $t \in \mathcal{N}$, and $\Gamma = (d)(((c_1)Q_1)\| \ldots \|((c_m)Q_m))$, $t \in [\![\Gamma]\!]$ iff $(0)(((c_1)Q_1)\| \ldots \|((c_m)Q_m)) \stackrel{*}{\Longrightarrow} (t-d)((0)\| \ldots \|(0))$.*

Proof : The backward direction is straightforward due to the soundness of the rewriting rules. For the forward direction, assume there is a rewriting sequence for Γ. We use induction on the lengths of all rewriting sequences. In the basic case, $(0)P \Longrightarrow (c)((0))$ and the case is proven. We assume that the lemma is true for all rewriting sequences of lengths $\leq h$. Now suppose we are given a rewriting sequence of length $h+1$. The following two case analyses complete the proof.

- Suppose at the beginning of the sequence, we have a rule like $P \rightarrow (c_1)P_1(c_2)$. According to the inductive hypothesis, $(0)P_1 \stackrel{*}{\Longrightarrow} (d_1)((0)\| \ldots \|(0))$ for some d_1. Thus $(0)P \stackrel{*}{\Longrightarrow} (c_1 + c_2 + d_1)((0)\| \ldots \|(0))$.
- Suppose at the beginning of the sequence, we have a rule like $P \rightarrow P_1 \| P_2$ and for the two child subtrees, we have $(0)P_1 \stackrel{*}{\Longrightarrow} (d_1)((0)\| \ldots \|(0))$ and $(0)P_2 \stackrel{*}{\Longrightarrow} (d_2)((0)\| \ldots \|(0))$ respectively. Then according to the semantics, $d_1 = d_2$. So we have $(0)P \stackrel{*}{\Longrightarrow} (d_1)((0)\| \ldots \|(0))$. ‖

Note that in determining the emptiness of something like $[\![(d)(((c_1)Q_1)\| \ldots \|((c_m)Q_m))]\!]$, the true value of d does not matter. Furthermore, any duplication of $(c_i)Q_i$ is as good as a single copy of $(c_i)Q_i$. Thus we propose to build an edge-weighted graph $\Delta_A = \langle U, W \rangle$ for PCAN $A = \langle \Pi, \eta, S, \Theta \rangle$ such that

- Each element in U is a nonempty subset of $\{(c)P \mid c \in \mathcal{N}; c \leq L_A; P \in \Pi\}$; and
- for each $v = \{((c_1)Q_1), \ldots, ((c_m)Q_m)\}, \hat{v} = \{((\hat{c}_1)\hat{Q}_1), \ldots, ((\hat{c}_n)\hat{Q}_n)\} \in U$ and $L_A \geq d \in \mathcal{N}$, $(v, \hat{v}) \in W$ is labelled with d iff $((c_1)Q_1)\| \ldots \|((c_m)Q_m) \Longrightarrow (d)(((\hat{c}_1)\hat{Q}_1)\| \ldots \|((\hat{c}_n)\hat{Q}_n))$.

Given a path $v_0 \stackrel{d_1}{\rightarrow} v_1 \stackrel{d_2}{\rightarrow} \ldots \stackrel{d_n}{\rightarrow} v_n$, the weight of the path is defined to be $d_1 + \ldots + d_n$. Thus the following theorem shows the correctness of the graph-construction.

THEOREM 7 *: Given a PCAN $A = \langle \Pi, \eta, S, \Theta \rangle$, with $\Delta_A = \langle U, W \rangle$, and $t \in \mathcal{N}$, $t \in [\![S]\!]_A$ iff there is a path of weight t in Δ_A from $\{(0)S\}$ to $\{(0)((0))\}$.*

Proof : Straightforward from definition of $\Longrightarrow$ and the construction of Δ_A. ‖

Time_Expression(A) /* $A = \langle \Pi, \eta, S, \Theta \rangle$ */ {
(1) Construct $\Delta_A = \langle U, W \rangle$.
(2) For each $v \in U$, {
  (1) For each $u \xrightarrow{\tau_1} v$, $v \xrightarrow{\tau_2} w$, $v \xrightarrow{\tau_3} v$, and $u \xrightarrow{\tau_4} w$, {
    (1) replace $u \xrightarrow{\tau_4} w$ by $u \xrightarrow{\tau_4 \cup (\tau_1 + \tau_2 + \tau_3 *)} w$ in W.
  }
}
(3) return the label of $((0)(S), (0)(0))$ in W.
}

Table 8: Algorithm for PCAN semilinear expressions

Furthermore, since Δ_A fully characterizes the pattern of process composition of A from S, we can use a Kleene's closure algorithm to calculate the semilinear expressions of $[\![A]\!]$ as in table 8. The number of nodes in Δ_A is $O(2^{|\Pi|} L_A^{|\Pi|}) = O((2L_A)^{|\Pi|})$. Since the Kleene's framework is of order three, the whole algorithm should take at most $O(((2L_A)^{|\Pi|})^3) = O((2L_A)^{3|\Pi|})$ time units.

5.2 Stratified CAN (SCAN)

SCAN is designed to be a superclass of PCAN and BCAN. In an SCAN A, conceptually, we have a sequence of PCAN's and BCAN's $A_1 = \langle \Pi_1, \eta_1, S_1, \Theta_1 \rangle, \ldots, A_n = \langle \Pi_n, \eta_n, S_n, \Theta_n \rangle$ such that

- for each $1 \leq i \leq n$, A_i is either BCAN or PCAN;
- for any $1 \leq i < j \leq n$, if $\Pi_i \cap \Pi_j \neq \emptyset$, then $\Pi_i \cap \Pi_j = \{S_j\}$ and S_j does not appear in the left-hand-side of any rules in Θ_i.

Formally speaking, SCAN $A = \langle \bigcup_{1 \leq i \leq n} \Pi_i, \bigcup_{1 \leq i \leq n} \eta_i, S_1, \bigcup_{1 \leq i \leq n} \Theta_i \rangle$.

The following example shows how the SCAN theory can be used to analyze the two well-accepted formalisms, statechart[9] and modechart[13].

Example 7 : Statechart is a graphical language for the description of computer systems[9]. Modechart is a real-time variation of statechart[13]. Both of them have been well-accepted and drawn much attention from industry and academia. In statechart and modechart, a system is represented by a *mode* and people may hierarchically construct their representations from smaller modes. Each mode is either *parallel* or *serial*. Transitions between modes can be triggered by timeouts or other message broadcasting.

For example, in figure 6, we have a three-layered system. Mode M is the parallel root mode with two child modes, M_1, M_2. Mode M_1 is a serial mode with three children, M_{1a}, M_{1b}, M_{1c} with M_{1a} as

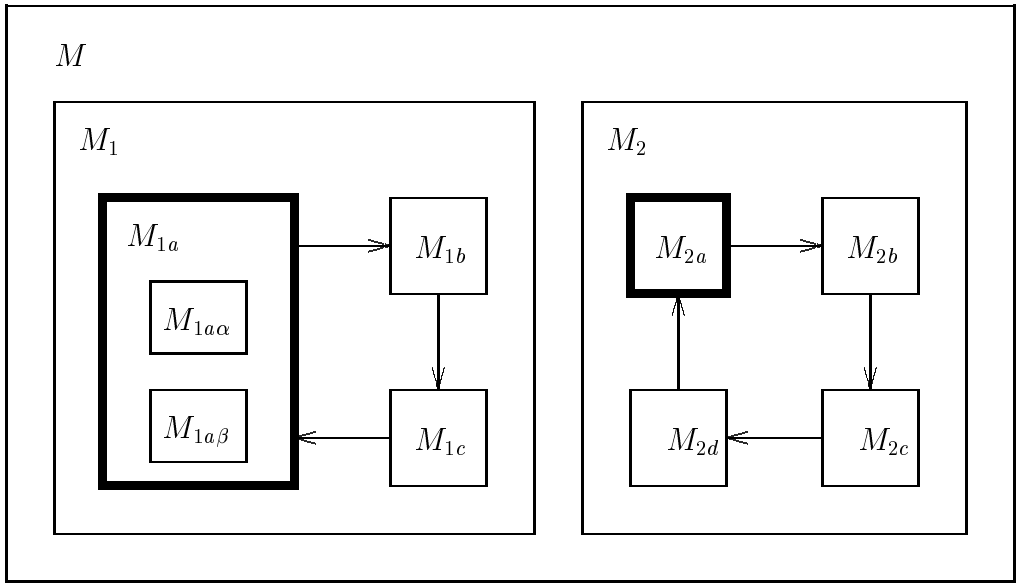

Figure 6: A statechart (modechart)

the starting child mode.

The modes with darkened boundaries are starting modes in serial modes. In the case that the parallel modes do not interact with each other until their exitting, we find that the corresponding analysis problems fall right in SCAN. ‖

We can show that SCAN is semilinear by the following lemma.

LEMMA 8 *: Given an SCAN representable by sequence $A_1, \ldots, A_n$ such that $A_i = \langle \Pi_i, \eta_i, S_i, \Theta_i \rangle$, for all $1 \leq i \leq n$, $\langle \bigcup_{i \leq j \leq n} \Pi_j, \bigcup_{i \leq j \leq n} \eta_j, S_i, \bigcup_{i \leq j \leq n} \Theta_j \rangle$ is semilinear.*

Proof : It is without doubt that A_n is itself semilinear. Now we want to show that A_i is semilinear by assuming that $\langle \bigcup_{j \leq k \leq n} \Pi_k, \bigcup_{j \leq k \leq n} \eta_k, S_j, \bigcup_{j \leq k \leq n} \Theta_k \rangle$ is semilinear for each $i < j \leq n$. Given a semilinear set of integers $\tau = \tau_1 \cup \tau_2 \cup \ldots \cup \tau_m$, there is a construction of $F_\tau \in \mathrm{BCAN} \cap \mathrm{PCAN}$ such that $[\![F_\tau]\!] = \tau$. The construction goes by working on each of $\tau_1, \ldots, \tau_m$ and then uniting the results together. Suppose for some $1 \leq h \leq m$, we have $\tau_h = \{a + b_1 j_1 + \ldots + b_k j_k \mid j_1, \ldots, j_k \in \mathcal{N}\}$ for some $a, b_1, \ldots, b_k \in \mathcal{N}$. It is easy to show that $[\![F_{\tau_h}]\!] = \tau_h$ with $F_{\tau_h} = \langle \Pi_{\tau_h}, \eta_{\tau_h}, S_{\tau_h}, \Theta_{\tau_h} \rangle$ such that

- $\Pi_{\tau_h} = \{S_{\tau_h}, D_{\tau_h}^{(1)}, D_{\tau_h}^{(2)}, \ldots, D_{\tau_h}^{(k)}\}$;
- $\eta_{\tau_h}(S_{\tau_h}) = \eta_{\tau_h}(D_{\tau_h}^{(1)}) = \eta_{\tau_h}(D_{\tau_h}^{(2)}) = \ldots \eta_{\tau_h}(D_{\tau_h}^{(k)}) = 0$;
- $\Theta_{\tau_h} = \{S_{\tau_h} \to (a) D_{\tau_h}^{(1)}\} \cup \{D_{\tau_h}^{(l)} \to (b_l) D_{\tau_h}^{(l)} \mid 1 \leq l \leq k\} \cup \{D_{\tau_h}^{(l)} \to D_{\tau_h}^{(l+1)} \mid 1 \leq l < k\} \cup \{D_{\tau_h}^{(k)} \to (0)\}$

Then $[\![F_L]\!] = \tau$ with $F_\tau = \langle \{S_\tau\} \cup \bigcup_{1 \leq h \leq m} \Pi_{\tau_h}, S_\tau, \{S_\tau \to S_{\tau_h} \mid 1 \leq h \leq m\} \cup \bigcup_{1 \leq h \leq m} \Theta_{\tau_h} \rangle$.

Now suppose we have established the semilinearity of $[\![S_{i+1}]\!]_A, [\![S_{i+2}]\!]_A, \ldots, [\![S_n]\!]_A$ and are waiting to show that of $[\![S_i]\!]_A$. Assume that we can use the just-mentioned construction to build $F_{[\![S_{i+1}]\!]}, F_{[\![S_{i+2}]\!]}, \ldots, F_{[\![S_n]\!]}$ for $[\![S_{i+1}]\!]_A, [\![S_{i+2}]\!]_A, \ldots, [\![S_n]\!]_A$ respectively. Furthermore, we assume that $F_{[\![S_j]\!]} = \langle \dot{\Pi}_j, \dot{\eta}_j, S_j, \dot{\Theta}_j \rangle$ for each $i < j \leq n$. The proof can be done by showing that $\langle \bigcup_{i \leq j \leq n} \Pi_j, \bigcup_{1 \leq j \leq n} \eta_j, S_i, \bigcup_{i \leq j \leq n} \Theta_j \rangle$ represents the same set of integers as $\langle \Pi_i \cup \bigcup_{i<j\leq n} \dot{\Pi}_j, \eta_i \cup \bigcup_{i<j\leq n} \dot{\eta}_j, S_i, \Theta_i \cup \bigcup_{i<j\leq n} \dot{\Theta}_j \rangle$. This is true because the stratified structure of $A_1, \ldots, A_n$ and the equivalence between $[\![S_j]\!]_A$ and $[\![F_{[\![S_j]\!]}]\!]$ for each $i < j \leq n$.

The proof then completes by noting that $\dot{A}_i$ is either BCAN or PCAN. ||

The above-mentioned proof for the semilinearity of SCAN also depicts an iterative algorithm to determine the emptiness of any given SCAN representation. Due to its straightforwardness, we shall omit the algorithm presentation here.

5.3 DRCAN : connections to recurrence equations

The third possibility, called *Deterministic Recurrence CAN (DRCAN)*, tries to connect to the rich theory of recurrence equation and allows some strict integer parametric synchronization between the child processes. Such a formulation can be useful in analyzing strongly heirarchical composition, like nested loops. In DRCAN, we may request each procedure invociation comes with a natural number parameter which tells how many more invocation levels there are to be done. A parallel rule does not increase the invocation height while a sequential one does. The syntax of DRCAN rules is

$$P^{\langle 0 \rangle} \to (c); \qquad P^{\langle k \rangle} \to P_1^{\langle k-1 \rangle} P_2^{\langle k-1 \rangle}; \qquad P^{\langle k \rangle} \to \max(P_1^{\langle k \rangle}, P_2^{\langle k \rangle}); \qquad P^{\langle k \rangle} \to \min(P_1^{\langle k \rangle}, P_2^{\langle k \rangle});$$

The three restriction we imposed on DRCAN for the simplicity of analysis are

- Recursion is only allowed when a procedure calls itself directly in a sequential rule.
- A procedure invocation (namely a nonterminal followed by a formal parameter) $P^{\langle h \rangle}$ with parameter $h \in \{0, k\}$ can only appear on the left-hand-side of one rule.
- From the left-hand-side to the right-hand-side of a rule, integer parameters cannot increment.

A typical example of using DRCAN in analyzing execution is nested loops.

Example 8 : Suppose we have the following nested loops.

```
(1)    for 1 ≤ i < n, do
(2)      for 1 ≤ j < i, do
(3)         wait for 3 time units
```

The system can be described by

$$P_1^{\langle k \rangle} \to P_1^{\langle k-1 \rangle} P_2^{\langle k-1 \rangle}; \qquad P_2^{\langle k \rangle} \to P_2^{\langle k-1 \rangle} P_3^{\langle k-1 \rangle}; \qquad P_3^{\langle k \rangle} \to P_3^{\langle k-1 \rangle}; \qquad P_3^{\langle 0 \rangle} \to (3);$$

||

It is easy to see that DRCAN is not semilinear. As for the execution time set, we found that due to the strict restriction on recursion, we can do it in two steps. First, we arrange the nonterminals in a sequence ordered by their invocation ordering. Then starting from the one that calls nobody else, we analyze one-by-one their execution time patterns as single variable polynomial functions.

Second, the only difference from the traditional recurrence equation analysis is that we have to take care of the max- and min-rules. Since the invocation sequence is deterministic, we can determine within which intervals a single-variable polynomial is greater than another. There will be only finite number of such intervals. So the outcome of a max-rule will be described as a set of polynomials with respect to a set of respective integer intervals.

6 Conclusion

We have developed the theories of CAN, PCAN, SCAN, and DRCAN which allow the execution time analysis of real-time specifications of nonregular real-time systems. We believe that the paradigms more or less correspond to the engineering designing wisdom of abstraction and information-hiding and feel hopeful that more insight to the issue of verifiability can be gained in the future along these lines.

References

[1] R. Alur, C. Courcoubetis, D.L. Dill. Model Checking for Real-Time Systems. 5th IEEE LICS, 1990.

[2] R. Alur, T.A. Henzinger, P.-H. Ho. Automatic Symbolic Verification of Embedded Systems. in Proceedings of 1993 IEEE Real-Time System Symposium.

[3] J.C.M. Baeten, J.A. Bergstra, J.W. Klop. Decidability of Bisimulation Equivalence for Process Generating Context-Free Languages. Tech. Rep. CS-R8632, 1987, CWI.

[4] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L.Dill, L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. 5th IEEE LICS, 1990.

[5] A. Bouajjani, R. Echahed, P. Habermehl. On the Verification Problem of Nonregular Properties for Nonregular Processes. 10th IEEE LICS, 1995.

[6] E. Clarke, E.A. Emerson, and A.P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. ACM TOPLAS 8(2), 1986, pp. 244-263.

[7] N. Chomsky. One Certain Formal Properties of Grammar. Information and Control, **2**:2, 137-167.

[8] S. Ginsburg, S.A. Greibach. Deterministic Context-Free Languages. Information and Control, **9**:6, 563-582.

[9] D. Harel. Statecharts : A Visual Formalism for Complex Systems." The Weizmann Institute of Science Technical Report, Israel (July 1986). Also in Science of Programming 8, 1987.

[10] T.A. Henzinger, Z. Manna, A. Pnueli. Temporal Proof Methodologies for Real-Time Systems. 18th ACM POPL, 1991.

[11] T.A. Henzinger, X. Nicollin, J. Sifakis, S. Yovine, Symbolic Model Checking for Real-Time Systems, 7th IEEE LICS, 1992.

[12] J.E. Hopcroft, J.D. Ullman. Introduction to Automata Theory, Languages, and Computation. Addison-Wesley, 1979.

[13] F. Jahanian, A.K. Mok. Modechart : A Specification Language for Real-Time Systems." to appear in IEEE Transactions on Software Engineering.

[14] X. Nicollin, J. Sifakis, S. Yovine. From ATP to Timed Graphs and Hybrid Systems. In Real-Time : Theory in Practice, LNCS 600, Springer-Verlag, 1991.

[15] R.J. Parikh. On Context-Free Languages. Journal of the Association for Computing Machinery, 4 (1966), 570-581.

[16] F. Wang, A.K. Mok, E.A. Emerson. Real-Time Distributed System Specification and Verification in APTL. ACM TOSEM, Vol. 2, No. 4, Octobor 1993, pp. 346-378. Also in 14th ACM ICSE, 1992.

[17] F. Wang. Timing Behavior Analysis for Real-Time Systems. 10th IEEE LICS, San Diego, 1995.

[18] F. Wang, C.T. Lo. Procedure-Level Verification of Real-Time Concurrent Systems. in Proceedings of the 3rd FME, Oxford, Britain, March 1996; LNCS 1051, Springer-Verlag.