International Journal of Foundations of Computer Science

Inductive Composition of Numbers with Maximum, Minimum, and Addition

*A New Theory for Program Execution-Time Analysis**

Farn Wang
Department of Electrical Engineering, National Taiwan University
Taipei, Taiwan 106, Republic of China
+886-2-23635251 ext. 435; FAX +886-2-23671909; farn@cc.ee.ntu.edu.tw

Received (received date)
Revised (revised date)
Communicated by Editor's name

ABSTRACT

We extend the classic work of R.J. Praikh on context-free languages with operators min and max on unary alphabet. The new theory is called *CAN (Compositional Algebra of Numbers)* and can be used to model software processes that can be concatenated, concurrently executed, and recursively invoked. We propose and analyze an algorithm which constructs the execution time sets of a CAN in semilinear form. Finally, we consider several interesting variations of CAN whose execution time sets can be constructed with algorithms.

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

1. Introduction

In the design of sophisticated systems, usually, various nonregular behaviors are described and need to be verified. Classic nonregular behaviors refer to those which cannot 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 research on formal methods and automated verification is booming some success reported everywhere, in general, there is still very little we can do to analyze such sophisticated systems. For example, in the model-checking approach of real-time systems, it is usually difficult to determine 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.*

*The work was partially supported by the NSC, Taiwan, ROC under grants NSC 90-2213-E-001-006 and NSC 90-2213-E-001-035, and the by the Broadband Network Protocol Verification Project of the Institute of Applied Science & Engineering Research, Academia Sinica, 2001.

In this paper, we propose a new theory for the execution time analysis of concurrent and recursive behaviors. We call the theory *Compositional Algebra of Numbers (CAN)* and show that it can help identify the behavioral regularity of a class of dynamically concurrent and recursive systems which may have previously been thought to be nonregular. Intuitvely, we interpret numerical operations as process constructors. Specifically, numerical addition represents the execution time of 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 that process S, with Boolean arguments x, y, can be fulfilled by first invoking a process P, with argument x, and after the fulfillment of $P\langle x\rangle$, invoking a process Q, with arguments $1, x, y$. Note here that x, y are unknown Boolean arguments at time of rule specification.

Parallel executions in CAN are modeled by 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 that process execution $S\langle x, y\rangle$ can be fulfilled through the simultaneous process invocations $P\langle x\rangle$ and $Q\langle 1, x, y\rangle$ and is fulfilled when either $P\langle x\rangle$ or $Q\langle 1, x, y\rangle$ 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 $Q\langle y\rangle$. An example of such parallel execution is parallel search in a distributed database.

Similarly, a rule like $S\langle x, y\rangle \rightarrow \max(P\langle x\rangle, Q\langle 1, x, y\rangle)$ means that process $S\langle x, y\rangle$ is fulfilled when both processes $P\langle x\rangle$ 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 larger of those of $P\langle x\rangle$ and $Q\langle 1, x, y\rangle$. An example of such parallel execution is parallel mergesort.

Given a CAN A, with a starting process type with Boolean constant arguments, 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 that describes the set. The main contribution of this work is to show, with a construction algorithm, that *for every CAN A, $[[A]]$ is seminlear.*[a] The result not only is of theorectical interest, but may also have potential in real-world applications. With this result, the highly recursive and dynamic (processes that can be dynamically instantiated and terminated) systems in CAN theory are now subject to algorithmic analysis. In the future, it can be interesting to see whether CAN theory can be used as a new weapon in our arsenal of verification technologies and can complement other well-established verification theories[2, 4, 5, 8, 13, 14, 29, 32, 33, 37, 48, 49, 53, 54] for recursive and parallel software systems.

We present this work as follows. In section 2, we compare our work with related works. In section 3, we formally define the syntax and semantics of CAN. In section 4, we derive an algorithm to compute the semilinear description of any given CAN. Variations of CAN theory can also be used to deal with various practical issues in system designs. In section 5, we consider CAN theory with divergence semantics (i.e., the assumption that a process can fail). Finally, in sections 6 and

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

7, we discuss two other possibilities for analyzing the numerical representations of high-level behaviors. The first one is called *PCAN (Parallel CAN)*, which applies rendezvous semantics to parallel operators while disallowing arbitrary concatenations. The second one, called *SCAN (Stratified CAN)*, allows finite alternation of concatenation and parallel rendezvous.

2. Related Work

We view CAN theory as a new and promising verification technique which can be used to complement, not compete with, the traditional verification theories and enrich our arsenal of verification techniques. In the following, we first discuss the relationship between our work and R. Parikh's classic work[44]. Then we discuss how our work can complement various well-established verification theories.

2.1. Relation with Parikh's Work

The classic work of R.J. Parikh on context-free languages[44] can be reinterpreted as a special case of our CAN. Assume we are given a language L with finite alphabet Σ in the 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 concerns semilinearity of integer vector sets. In our framework, we only care about the case with a unary alphabet. Given $w \in L$ with a 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*[b] of classical process algebra and represents a truly context-free fragment of CAN since it lacks a parallel operator. Thus we also refer to Parikh's subclass (with unary alphabet) of CAN as *Basic CAN (BCAN).* In other words, BCAN is a subclass of CAN without max and min-rules. We envision the possibility of extending Parikh's work with parallel operators to enhance our ability to specify and verify the timing aspects of nonregular real-time systems.

Compared with classical context-free language with general alphabets[15], our work may enable the analysis of real-time concurrent systems that might have previously been thought to not be analyzable. For one thing, in the general case, the emptiness problem of classical context-free language intersection is undecidable[25] while in sections 4, 6, and 7, we shall see that under the CAN, PCAN, and SCAN paradigms, this problem becomes decidable and may be used to analyze process execution time compositions.

2.2. Comparison with Various Theories

In the computation of concurrent systems, there are two major factors that

[b]BPA stands for *Basic Process Algebra*, the fragment of process algebra without parallel compositions.

control the system behavior: communication and time passage[31, 43]. In the traditional theories of formal verification[2, 5, 8, 13, 14, 32, 53], 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[3, 7]. Still, very little has been done to exploit specification structures and engineering disciplines 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. These processes are called numerical because from the outside, we can only observe their full execution times. Thus, to some extent, the 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. We believe that the CAN theory deserves further investigation and may lead to new techniques for the analysis of dynamic and recursive systems.

In the following, we compare with specific related work in the literature.

2.2.1. Parametric analysis of dense systems

In this analytical framework, we have a (timed or hybrid) automata and a specification formula with unknown parameters. The values of parameters do not change with time. A valuation of parameters that makes the specification satisfied is called a *solution*. The framework aims to compute the characterization of solutions. In [4], Alur, Henzinger, and Vardi showed that for timed automatas with three or more timing parameter variables, the emptiness problem of solution space is undecidable. In [3], Alur, Henzinge, and Ho presented a symbolic model-checking procedure for linear hybrid systems, based on manipulation algorithms of convex polyhedra. The procedure does not guarantee termination since reachability problem of linear hybrid systems is undecidable.

Wang discovered that if the parameter variables are restricted to the specification formulas in *parametric timed CTL*, the semilinear expressions to characterize solution spaces can be constructed with an algorithm[48]. The technique is based on Kleene closure algorithm for calculating the semilinear expressions for timing distances between regions of timed automata. In 1997, Wang also extended his algorithm to handle frameworks with static parameters (which are not compared with clocks) in timed automatas[49]. In 2001, Wang and Yen again extended the algorithm for an integrated framework for both controller synthesis and parametric analysis[54].

In 2003, Wang and Yen investigated the solution characterization complexities of timed automata with restrictions on parameters[55]. A parameter is called an *unpperbound (lowerbound)* parameter if it is used as an upperbound (or lowerbound respectively) to compare with a clock in timed automata. A timed automata is an upperbound (lowerbound) automata if all its parameter variables are upperbounds (or lowerbounds respectively). A timed automata is *bipartite* if none of its parameter variables is both an upperbound and a lowerbound. It is shown in [55] that the solution characterizations of upperbound timed automata are semilinear and of double exponential complexity, those of lowerbound timed automata are semilinear and of single exponential complexity, and those of bipartite timed automata are incomputable.

However, such analytical techniques do not allow us to naturally model the execution times of general recursive softwares since timed and hybrid automatas are not recursive. CAN theories can potentially enrich these results and help extend those works to recursive and dynamic software systems. For example, it is imaginable that we can introduce atomic wait-statements into these frameworks such that the waiting time is characterized by CAN theory. This paradigm is actually quite compatible with our SCAN theory which we describe in section 7.

2.2.2. Parametric Analysis of Counter Systems

A counter is a mathematical device that can be incremented, decremented, and tested against zero. The states of counter machines record the vectors of counter values. Conceptually, such counters can be used to encode the push-down stacks in recursive procedure-calls. The characterizations of counter machine state-spaces are incomputable since two-counter machine halting problem is undecidable[27, 34]. But Ibarra et al have shown that when only one non-reversal-bounded push-down stack is used, the emptiness, reachability, non-safety, and invariance problems can all be solved with algorithms[35]. Various adaptations of this technique can be applied to systems with various mathematical storage devices. For example, a recent work by Dang et al characterizes the numbers of times some transitions are executed in a finite-state system[56].

2.2.3. Logics and Arithmetics

People have also used various fragments of first-order and higher-order logics for the formal description and analysis of timing behaviors of real-time systems[10, 36]. Great expressiveness for non-regular behaviors comes with such logics. Consequently, computability for solution space characterizations is impossible with such non-regular behaviors. Many semi-decision procedures and theorem-provers have been developed to help automating the analysis of systems modeled in such logics[19, 42].

A well-discussed class of first-order arithmetics is *Presburger arithmetic*, which characterizes semilinear sets of integers. A Presburger arithmetic formula is composed of linear constraints like $a_1x_1 + \ldots + a_nx_n \leq c$, Boolean operators, and quantifications on integer variables. It falls in the well-known WS1S (Weak Second-order logic with 1 Successor) and is known to subject to algorithmic analysis[6, 23, 24, 26, 30, 38, 46, 51]. Many tools use finite-state automata to represent and manipulate semilinear sets of integer vectors. Algorithm for concatenation, unions, intersections, complements, and Kleene's closure have also been presented. Efficient library routines for the manipulations of Presburger arithmetics can be found in Omega library[45]. Porject MONA has also developed various tools to analyze models in

WS1S based on automata manipulation[30].

However, there is no immediate syntax structures in WS1S and Presburger arithmetics for general software recursions for concurrent softwares. In comparison, there is no restriction on recursion depth and child processes can be created and ended dynamically in our CAN theory. It may be interesting to see how our techniques to transform CAN formulas to ones without max() and min() operators will serve as inspiration for people to enrich the syntax of Presburger arithmetic without sacrificing its algorithmic analyzability.

2.2.4. Process Algebra

CAN is similar in its syntactical structure to classical process algebra theory[5, 8, 33]. Thus, it is possible to treat CAN as a new semantic definition of process algebra (as long as we know the execution times of atomic events). However, in classical process algebra, parallel composition is defined based on interleaving semantics, and there is really nothing similar to our parallel operator, min() and max(), with a kind of true-concurrency flavor. Our parallel composition semantics for CAN provides a new viewpoint toward the execution-time analysis of complex real-time concurrent systems, in which time-progress in concurrent components overlaps rather than interleaves.

There also have been numerous works to extend process algebra with real-time clocks. ACSR is an extenstion for the specification of resource-contentions in real-time sytems[17]. The main analysis method in ACSR is state-space exploration, for finite-state systems, and simulation, for general systems. Baeten and Middelburg have extended BPA and ACP with discrete and dense time clocks[11]. Decidabilities of timed extensions of *BPP*s[c] have also been discussed[9, 39].

Our work does not compete with the classical and timed process algebrae. For example, the classical and timed ones usually have extensive capability to describe synchronizations among processes. But they usually have very few algorithmic supports in analyzing general parallel recursions. On the other hand, our CAN theory simplifies the capabilities in synchronizations in exchange for the algorithmic analysis of general parallel recursions. Moreover, our parallel compositions support the estimation of minimum and maximum execution times of parallel processes. But in classical and timed extensions of process algebrae, there is no direct counterparts in this regard. Thus it is interesting to see how our techniques can be used to enhance the analysis power of process algebra tools.

2.2.5. Constraint-Solving

Traditionally, engineers have also used various constraint systems to characterize synthesis and optimization tasks in the industry. Many classic techniques, e.g. Gaussian elimination, Simplex methods[16, 28, 40], and double dscription method[22] can then be applied on linear systems for engineering solutions to in-

[c]BPP stands for *Basic Parallel Processes*, the fragment of process algebra with sequential concatenations restricted to the form of $P \rightarrow aQ$, where a is an atomic event

dustry projects. For non-linear systems, techniques like factorizing, root solving, and cylindrical algebraic decomposition can be employed[12, 18].

In such a framework, there is no straightforward support for recursion and parallelism. One way to model real-time systems is to use uninterpreted function symbols, e.g., as event occurrence times[36]. But constraint systems with uninterpreted function symbols in general result in verification undecidability[20, 52]. In contrast, CAN allows for a natural presentation of procedural structures with composition rules for concatenation, concurrency, and recursion and has algorithm to construct the corresponding semantics in semilinear expressions.

2.2.6. Worst-case Execution Time Analysis

Finally, the research on WCET (Worst-Case Execution Time) analysis has the same goal as our CAN theory. For instance, in [1], Audsley et al discusses how to calculate the WCET for a set of periodic and sporadic tasks with a fixed priority preemptive scheduler.

Meyer and Wong-Toi has also worked on the WCET of acyclic process precedence graphs, in which conjunctive precedence relations among processes are specified[41]. The WCET is analyzed for fixed priority dynamic schedulers.

In [21], techniques for detailed analysis of execution times of program implementations. Thus, the cycle times of machine instructions and hardware features like cache lookahead and interrupt handling all have to be taken into consideration.

In contrast to WCET research, Our CAN theories not only deduce the upper-bounds of execution times but also construct the characterizations of all execution times. Moreover, previous WCET research focuses on non-recursive systems. In this regard, our theories can also potentially complement WCET research.

3. CAN

3.1. Syntax

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 arguments, we shall let $\texttt{lhs}(\theta) = P\langle X\rangle$, i.e., the left-hand-side of rule θ is $P\langle X\rangle$.

A CAN A is a tuple $\langle \Pi, \eta, P_0\langle X_0\rangle, \Theta\rangle$, where Π is a finite set of process names, $\eta : \Pi \mapsto \mathcal{N}$ defines the number of arguments received by each process, $P_0\langle X_0\rangle$ is the starting process and initial Boolean argument values, and Θ is a set of process construction rules. For each $P \in \Pi$, $\eta(P)$ is called the *arity* of process type P. The rules in Θ are in one of the following four forms:

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

Here X, X_1, and X_2 are finite sequences of Boolean arguments, which can be either Boolean variables or Boolean constants. Specifically, we shall call such a sequence an *argument sequence* and such a Boolean variable an *argument variable*. Intuitively, these Boolean arguments are like the input arguments to functional programming procedure-calls (e.g. Prolog). When the arguments are constants, the execution of the procedure-call may use the constants as input parameters. When some arguments are variables, the execution of the procedure-call derives the restriction on the argument variables.

Semicolons are used as separators between rule presentations and are not part of the rules. 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 as follows.

- Rule $P\langle X\rangle \rightarrow (c)$ means that process $P\langle X\rangle$ (i.e., process P with arguments 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 $P\langle X\rangle$ can be executed by first executing a process $P_1\langle X_1\rangle$ and then after the completion of $P_1\langle X_1\rangle$, immediately executing a process $P_2\langle X_2\rangle$.
- Rule $P\langle X\rangle \rightarrow \max(P_1\langle X_1\rangle, P_2\langle X_2\rangle)$ means that process $P\langle X\rangle$ can be executed by executing in parallel a process $P_1\langle X_1\rangle$ and a process $P_2\langle X_2\rangle$, and that execution is completed when both $P_1\langle X_1\rangle$ and $P_2\langle X_2\rangle$ are completed.
- Rule $P\langle X\rangle \rightarrow \min(P_1\langle X_1\rangle, P_2\langle X_2\rangle)$ means that process $P\langle X\rangle$ can be executed by executing in parallel a process $P_1\langle X_1\rangle$ and a process $P_2\langle X_2\rangle$, and that execution is completed when either $P_1\langle X_1\rangle$ or $P_2\langle X_2\rangle$ is completed.

We require that in any given CAN $\langle\Pi, \eta, P_0\langle X_0\rangle, \Theta\rangle$, 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 arguments must be consistent with its process' arity.

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

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

$S\langle x, 0\rangle \rightarrow (3)$;	$S\langle x, y\rangle \rightarrow S\langle x, 0\rangle S\langle 0, y\rangle$;
$S\langle x, y\rangle \rightarrow P\langle x\rangle Q\langle 1, x, y\rangle$;	$S\langle x, y\rangle \rightarrow \min(Q\langle x, y, 1\rangle, S\langle y, 0\rangle)$;
$S\langle x, y\rangle \rightarrow \max(P\langle x\rangle, S\langle 0, y\rangle)$;	$P\langle x\rangle \rightarrow (4)$;
$Q\langle x, y, 1\rangle \rightarrow (6)$;	

The first rule says that an $S\langle x, 0\rangle$ process (S process with arguments $x, 0$) 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 can be the concatenation of a $P\langle x\rangle$ process and a $Q\langle 1, x, y\rangle$ process, or can be the parallel execution of a $Q\langle x, y, 1\rangle$ process 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 $P\langle x\rangle$ process and an $S\langle 0, y\rangle$ process (fulfilled when both $P\langle x\rangle$ and the child $S\langle 0, y\rangle$ are fulfilled). □

3.2. Formal Semantics

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 that 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.

Suppose we are given a CAN $A = \langle \Pi, \eta, P_0\langle X_0\rangle, \Theta\rangle$. An *interpretation* $\mathcal{I}$ for A is a mapping from $\{x \mid x \text{ is an argument variable in } A\} \cup \{0,1\}$ to $\{0,1\}$ such that $\mathcal{I}(0) = 0$ and $\mathcal{I}(1) = 1$. Given an interpretation $\mathcal{I}$ and an argument sequence X, $\mathcal{I}(X)$ is an argument constant sequence obtained from X by substituting $\mathcal{I}(x)$ for every argument variable $x \in X$.

The following concept formalizes the mechanism used to copy input argument values to the formal arguments in process declarations. Given interpretations $\mathcal{I}, \mathcal{I}'$ for A and argument sequences X, X' for process type P, $\mathtt{pmatch}_{P\langle X\rangle \to \langle X'\rangle}(\mathcal{I}, \mathcal{I}')$ intuitively defines the relation that in a process invocation of P with actual arguments $\langle X\rangle$ in the caller and formal argument places $\langle X'\rangle$ in the callee,

- $\mathcal{I}$ represents an interpretation of actual argument values consistent with the argument values in X from which we copy to X'; and
- $\mathcal{I}'$ represents an interpretation of formal argument values consistent with the values copied to arguments in X'.

The relation is formally defined in the following way. Suppose $X = x_1, \ldots, x_n$ and $X' = x'_1, \ldots, x'_n$. $\mathtt{pmatch}_{P\langle X\rangle \to \langle X'\rangle}(\mathcal{I}, \mathcal{I}')$ is *true* iff X is consistent with X' element by element; i.e., for all $1 \le i \le n$, $\mathcal{I}'(x'_i) = \mathcal{I}(x_i)$. This definition of $\mathtt{pmatch}()$ allows us flexibility to variable-sharing among child process invocations. For example, we may call process P with actual arguments $\langle 1, 0\rangle$ through the rule $P\langle x, y\rangle \to \max(P_1\langle x, z\rangle, P_2\langle z, y\rangle)$. In this situation, x and y will be instantiated to 1 and 0, respectively, while z can be instantiated to either 0 or 1. Either of the instantiations satisfies the requirement of $\mathtt{pmatch}()$. Intuitively, this means that z is a variable used for communication between child processes P_1 and P_2.

In the following, we modify the derivation trees of context-free languages to formally define the semantics of CAN.

Definition 1 *:* <u>*Execution trees*</u> *Given a CAN* $A = \langle \Pi, \eta, P_0\langle X_0\rangle, \Theta\rangle$*, a* $P' \in \Pi$*, and an argument sequence* X'*, an* execution tree $T = \langle V, E, r, \mu, \nu\rangle$ *for* $P'\langle X'\rangle$ *is a nonempty finite labeled tree such that* V *is the set of nodes in* T*,* E *is the set of arcs in* T*,* $P_0\langle X_0\rangle$ *is the* root *of* T*,* μ *is a partial mapping from* V *to* Θ*, and* ν *is a partial mapping from* V *to the set of interpretations for* A*. In addition, the following three requirements are satisfied.*

- *For each internal node* $v \in V$ *with left and right children* u_1 *and* u_2*, respectively, in* T*,* $\mu(v)$ *is in the form of* $P\langle X\rangle \to \max(P_1\langle X_1\rangle, P_2\langle X_2\rangle)$*,* $P\langle X\rangle \to \min(P_1\langle X_1\rangle, P_2\langle X_2\rangle)$*, or* $P\langle X\rangle \to P_1\langle X_1\rangle P_2\langle X_2\rangle$ *such that* $\mathtt{lhs}(\mu(u_1))$ *is in the form of* $P_1\langle X'_1\rangle$ *with* $\mathtt{pmatch}_{P_1\langle X_1\rangle \to \langle X'_1\rangle}(\nu(v), \nu(u_1))$*; and* $\mathtt{lhs}(\mu(u_2))$ *is in the form of* $P_2\langle X'_2\rangle$ *with* $\mathtt{pmatch}_{P_2\langle X_2\rangle \to \langle X'_2\rangle}(\nu(v), \nu(u_2))$*.*
- *For each leaf* $v \in V$*,* $\mu(v)$ *is in the form of* $P\langle X\rangle \to (c)$*.*
- $\mathtt{lhs}(\mu(r)) = P'\langle (X')\rangle$*.* $\square$

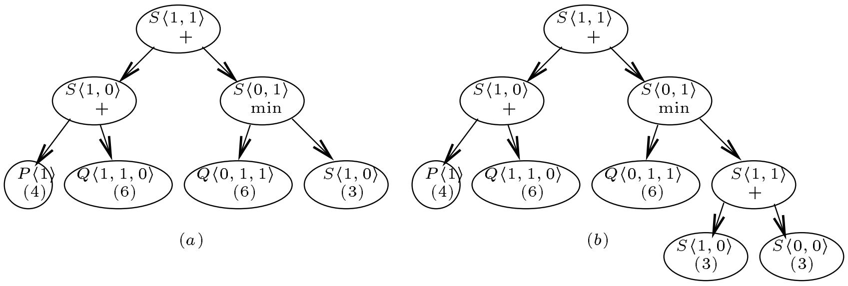


Fig. 1. Two execution trees

Note that we require execution trees to be finite labeled trees in the above. This prevents us from handling the chaotic behaviors of nonterminating recursions or undeclared procedure-calls. In section 5, we present a formal treatment of this issue. Also, since each internal node is labeled with a rule, the number of subtrees of the node is implicitly determined by the corresponding rule. For convenience of illustrating execution trees in the rest of the manuscript, 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 $\mathcal{I}(x)$ for every argument variable x in θ.

Example 2 *: For the CAN given in example 1, we may have the two execution trees shown in figure 1. In each node v, we only label* $\mathtt{lhs}(\mu(v)^{\nu(v)})$*, the left-hand-side process type with instantiated arguments, and the operator of $\mu(v)$ (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 instantiated construction of the relevant processes.* □

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

- 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 one of the following three conditions is true: (1) $\mu(r)$ is a sequential rule and $[[T]] = [[T_1]] + [[T_2]]$; (2) $\mu(r)$ is a min-rule and $[[T]] = \min([[T_1]], [[T_2]])$; and (3) $\mu(r)$ is a max-rule and $[[T]] = \max([[T_1]], [[T_2]])$.

For convenience, from now on, internal nodes labeled with addition (sequentional) rules, minimum rules, and maximum rules will respectively be 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 1, the execution times are 13 and 16 respectively.* □

Now the semantics of CAN are defined as the set of execution times of the starting process invocation.

Definition 2 *: Semantics of CAN Given a CAN $A = \langle \Pi, \eta, P_0\langle X_0\rangle, \Theta\rangle$, for each $P \in \Pi$ and argument sequence X such that $|X| = \eta(P)$, we let*

$$[[P\langle X\rangle]]_A = \{[[T]] \mid T \textit{ is an execution tree for } P\langle X\rangle \textit{ in } A\}$$

Process $P\langle X\rangle$ is said to have an execution if $[[P\langle Y\rangle]]_A \neq \emptyset$. The semantics of A, denoted as $[[A]]$, is defined as $[[P_0\langle X_0\rangle]]_A$. □

4. Deriving Semilinear Representations of CAN

The algorithm used to generate the semilinear description is presented and proved in three steps. First in section 4.1, we show that CAN is equivalent to CAN without arguments. In section 4.2, we explain our idea for eliminating max-rules and we rigorously establish that CAN is equivalent to CAN without max-rules. In section 4.3, we explain our idea for eliminating min-rules based on knowledge about which process types have unbounded execution times, we then present an algorithm that can tell if the execution time set of a CAN is of finite size. We then rigorously establish that CAN is equivalent to CAN without min-rules.

4.1. Removing Arguments

For a fixed number of Boolean argument positions, we can compute all the Boolean argument value combinations.

Given a CAN $A = \langle \Pi, \eta, P_0\langle X_0\rangle, \Theta\rangle$, we generate $\check{A} = \langle \check{\Pi}, \check{\eta}, S, \check{\Theta}\rangle$ such that $\check{\Pi} = \{S\} \cup \{P^\chi \mid P \in \Pi; \chi \in \{0,1\}^{\eta(P)}\}$, $\forall P^\chi \in \check{\Pi}(\check{\eta}(P^\chi) = 0)$, and

$$
\begin{aligned}
\check{\Theta} = \quad & \left\{ P_0 \to P_0^{\mathcal{I}(X_0)} \,|\, \mathcal{I} \text{ is a Boolean valuation to arguments in } A. \right\} & (1)\\
\cup \quad & \left\{ P^{\mathcal{I}(X)} \to (c) \;\middle|\; \begin{array}{l} P\langle X\rangle \to (c) \in \Theta; \mathcal{I} \text{ is a Boolean} \\ \text{valuation to arguments in } A. \end{array} \right\} & (2)\\
\cup \quad & \left\{ P^{\mathcal{I}(X)} \to P_1^{\mathcal{I}(X_1)} P_2^{\mathcal{I}(X_2)} \;\middle|\; \begin{array}{l} P\langle X\rangle \to P_1\langle X_1\rangle P_2\langle X_2\rangle \in \Theta; \\ \mathcal{I} \text{ is a Boolean valuation} \\ \text{to arguments in } A. \end{array} \right\} & (3)\\
\cup \quad & \left\{ P^{\mathcal{I}(X)} \to \max(P_1^{\mathcal{I}(X_1)}, P_2^{\mathcal{I}(X_2)}) \;\middle|\; \begin{array}{l} P\langle X\rangle \to \max(P_1\langle X_1\rangle, P_2\langle X_2\rangle) \in \Theta; \\ \mathcal{I} \text{ is a Boolean valuation} \\ \text{to arguments in } A. \end{array} \right\} & (4)\\
\cup \quad & \left\{ P^{\mathcal{I}(X)} \to \min(P_1^{\mathcal{I}(X_1)}, P_2^{\mathcal{I}(X_2)}) \;\middle|\; \begin{array}{l} P\langle X\rangle \to \min(P_1\langle X_1\rangle, P_2\langle X_2\rangle) \in \Theta; \\ \mathcal{I} \text{ is a Boolean valuation} \\ \text{to arguments in } A. \end{array} \right\} & (5)
\end{aligned}
$$

The union component in line (1) represents the consideration of all possible instantiations to argument variables in X_0. The union components in lines (2) through (5) represent instantiations of all rules in A. The requirement enforced by `pmatch()` is now fulfilled through enumeration of all possible interpretations. The following lemma shows that $[[A]] = [[\check{A}]]$.

Lemma 1 *Given a CAN $A = \langle \Pi, \eta, P_0\langle X_0\rangle, \Theta\rangle$, for every execution tree $T = \langle V, E, r, \mu, \nu\rangle$ for A, there is an isomorphic execution tree $\check{T} = \langle \check{V}, \check{E}, \check{r}, \check{\mu}, \check{\nu}\rangle$ for $\check{A}$ such that for every node v in T and its corresponding node $\check{v}$ in $\check{T}$, if $\mu(v) = P\langle X\rangle \to P_1\langle X_1\rangle P_2\langle X_2\rangle$ (or if $P\langle X\rangle \to \max(P_1\langle X_1\rangle, P_2\langle X_2\rangle)$ or $P\langle X\rangle \to \min(P_1\langle X_1\rangle, P_2\langle X_2\rangle)$), then $\check{\mu}(\check{v}) = P^{\nu(X)} \to P_1^{\nu(X_1)} P_2^{\nu(X_2)}$ (or $P^{\nu(X)} \to \max(P_1^{\nu(X_1)}, P_2^{\nu(X_2)})$ or $P^{\nu(X)} \to \min(P_1^{\nu(X_1)}, P_2^{\nu(X_2)})$), respectively.*

Proof. The isomorphism between the structures of A and $\check{A}$ is straightforward from the definition of the $\check{\Theta}$ and $\check{\mu}()$. Specifically, the definition of $\check{\mu}()$ incorporates the requirements of both $\mu()$ and $\nu()$. □

4.2. Removing max*-rules*

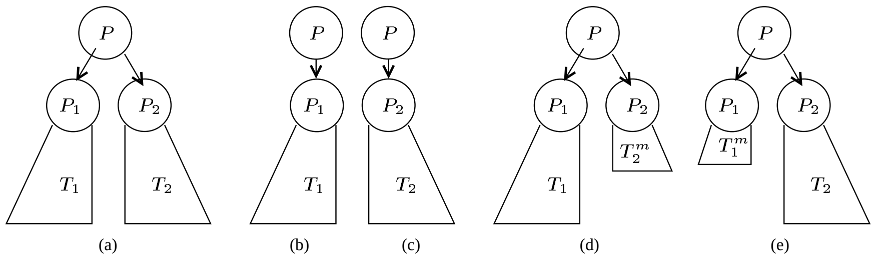


Fig. 2. Illustration of how to eliminate max-rules

From now on, we shall assume that all CANs given to us have no arguments. We first give an intuitive explanation of our reduction from CANs with max-rules to CANs without them. Then we present the formal definitions and lemmas.

The idea for eliminating max-rules is as follows. Given a max-rule like $P \to \max(P_1, P_2)$, we want to find out in what situation, the rule can be replaced by the following two rules: $P \to P_1$ and $P \to P_2$. Or equivalently, we want to find out in what situation, rule $P \to \max(P_1, P_2)$ helps generate the same set of execution times as rules $P \to P_1$ and $P \to P_2$ do. Note that the set of execution times that can be constructed with rule $P \to \max(P_1, P_2)$ is always contained in the set of execution times that can be constructed with the two replacements. The reason can be seen in figure 2. Consider the execution tree shown in figure 2(a). Before replacement, the execution time of only one child subtree in figure 2(a) will definitely be included in the semantics of P. But after replacement, we have the two execution trees in figure 2(b) and (c). Therefore, the execution times of both child subtrees in figure 2(a) will be included in the semantics of P.

Thus the applicability of the replacement depends on the following question: *"Under what condition, does the set of execution times that can be constructed with rule* $P \to \max(P_1, P_2)$; *always contain the set of execution times that can be constructed with the two replacements ?"* One sufficient condition for replacement is that $[[T_1]]$ is no smaller than the minimum execution time of P_2 and $[[T_2]]$ is no smaller than the minimum execution time of P_1 either. Assume that T_1^m and T_2^m are execution trees for P_1 and P_2, respectively, with minimum execution times. In this situation, the execution tree in figure 2(b) is equivalent to the one in figure 2(d) as far as execution time is concerned. (The same is true for figure 2(c) to figure 2(e).) This verifies our intuition that max-rules can be removed. However, the key element in the transformation is the computation of the minimum execution time, if any, of each process type.

4.2.1. Finding the Minimum Execution Times

The following procedure returns the minimum execution times in m_P for all $P \in \Pi$.

---

int m_P for each $P \in \Pi$;
`min_time`(A) /* $A = \langle \Pi, \eta, P_0, \Theta \rangle$ */ {

for each $P \in \Pi$, let $m_P := \infty$;
repeat until no more changes are possible, (1)
for each $\theta \in \Theta$,
if θ is like $P \rightarrow (c)$, let $m_P := \min(m_P, c)$; (2)
else if θ is like $P \rightarrow P_1 P_2$, let $m_P := \min(m_P, m_{P_1} + m_{P_2})$;
else if θ is like $P \rightarrow \max(P_1, P_2)$, let $m_P := \min(m_P, \max(m_{P_1}, m_{P_2}))$;
else if θ is like $P \rightarrow \min(P_1, P_2)$, let $m_P := \min(m_P, m_{P_1}, m_{P_2})$;
return m_P for all $P \in \Pi$;
}

In each iteration of the repeat-cycle at statement (1), we try to reduce the m_Ps' values by applying the rules to the m_Ps' values.

Given a CAN A and a process type P in A, we let $\min[[P]]_A$ be the minimum execution time of process P in A. Conveniently, when $[[P]]_A = \emptyset$, we let $\min[[P]]_A = \infty$. Now we have to prove that procedure `min_time()` eventually returns with $m_P = \min[[P]]_A$ for all $P \in \Pi$. This can be done with the help of the following lemma.

Lemma 2 *Given a CAN A, a process type P, and $h \in \mathcal{N}$, at the end of the h-th iteration of the repeat-cycle at statement (1) in `min_time()`, m_P is no greater than any execution time of the execution tree for P with height $\leq h$.*

Proof. Note that m_P is always assigned expressions like "$\min(m_P, \ldots)$." This means that the value of m_P will never increase in the repeat-cycle. The proof can be done by induction on h. In the induction step, according to the inductive hypothesis, we know that the lemma is true for the child subtrees of T. Thus the lemma is true at $h+1$ according to the semantics of the three rule types of $P \rightarrow P_1 P_2$, $P \rightarrow \max(P_1, P_2)$, and $P \rightarrow \min(P_1, P_2)$. □

A little more observation on the lemma can be made. The proof of Lemma 2 holds even if the repeat loop were an infinite loop, and therefore holds for all trees of finite heights. Furthermore, when some m_P is changed, it is decreased to a natural number. Because this can only happen finitely many times, and because no change can possibly occur once an iteration results in no change, the procedure must terminate with m_P's recording the minimum execution times.

4.2.2. Transformation into CANs without max-rules

Now with the knowledge of $\min[[P]]_A$, we can explain how to eliminate max-rules. We use the special symbol $c^{\ll}$ to denote any integer no smaller than c. For example, $5^{\ll}$ represents any integer no smaller than 5. We can define the relations of addition with bound c over elements in $\{0, \ldots, c-1, c^{\ll}\}$ as follows:

$$i +^{\langle c \rangle} j = \begin{cases} i + j & \text{if } i + j < c \wedge i \neq c^{\ll} \wedge j \neq c^{\ll}; \\ c^{\ll} & \text{otherwise.} \end{cases}$$

For example, $3 +^{\langle 5 \rangle} 4 = 5^{\ll}$ and $3 +^{\langle 5 \rangle} 1 = 4$. Intuitively, $d \leq c^{\ll}$ iff $d \leq c$. Let $M_A = \max\{\min[[P]]_A | P \in \Pi; \min[[P]]_A \neq \infty\}$. We can replace max-rules with the following technique of bounded execution time composition pattern enumeration.

1. For each process type P, we replace it with *expanded process types* $P^{[0]}, P^{[1]}, \ldots, P^{[M_A - 1]}, P^{[M_A^{\ll}]}$, where for each $0 \leq c < M_A$, $P^{[c]}$ means pro-

cess type P with execution time of exactly c; and $P^{[M_A^{\ll}]}$ represents process type P with execution time no less than M_A.

2. For each rule like $P \to \max(Q, R)$;, we replace it with enumeration rules of the following four groups:

$$\begin{array}{ll} P^{[c]} \to Q^{[c]}, & \text{for all } \min[[R]]_A \le c < M_A \\ P^{[c]} \to R^{[c]}, & \text{for all } \min[[Q]]_A \le c < M_A \\ P^{[M_A^{\ll}]} \to Q^{[M_A^{\ll}]}, & \min[[R]]_A \ne \infty \\ P^{[M_A^{\ll}]} \to R^{[M_A^{\ll}]}, & \min[[Q]]_A \ne \infty \end{array}$$

   Rules like $P^{[c]} \to Q^{[c]}$ with $c < \min[[R]]_A$; are not included since they do not generate an execution time that can be constructed with $P \to \max(Q, R)$. An enumeration rule like $P^{[c]} \to Q^{[c]}$ or $P^{[M_A^{\ll}]} \to Q^{[M_A^{\ll}]}$ generates an execution time in the original semantics when $c \ge \min[[R]]_A$. Intuitively, the replacement enumerates all possible composition patterns with execution times less than M_A. When the execution time is no less than M_A, the composition pattern behaves as if no max-rules exist. Of course, the replacement can be done only if there exist execution trees of Q and R.

3. For the other rule types, we enumerate all the composition patterns with expanded process types. For example, for a rule like $P \to QR$, we enumerate rules $P^{[c]} \to Q^{[c_1]}R^{[c_2]}$ for all $c = c_1 +^{\langle M_A \rangle} c_2$. Also, for a rule like $P \to \min(Q, R)$, we enumerate rules $P^{[c]} \to \min(Q^{[c_1]}, R^{[c_2]})$ for all $c = \min(c_1, c_2)$ with $c, c_1, c_2 \in \{0, 1, \ldots, M_A - 1, M_A^{\ll}\}$.

Given a CAN $A = \langle \Pi, \eta, P_0, \Theta \rangle$ (note that P_0 has no arguments, and that $\langle\rangle$ is omitted), we shall derive $\bar{A} = \langle \bar{\Pi}, \bar{\eta}, P_0, \bar{\Theta} \rangle$ such that there is no max-rule in $\bar{\Theta}$ and $[[\bar{A}]] = [[A]]$. The new CAN $\bar{A} = \langle \bar{\Pi}, \bar{\eta}, P_0, \bar{\Theta} \rangle$ is defined as follows:

$$\bar{\Pi} \stackrel{def}{=} \Pi \cup \{P^{[i]} \mid P \in \Pi; i \in \mathcal{N}; 0 \le i < M_A\} \cup \{P^{[M_A^{\ll}]} \mid P \in \Pi\},$$

where $\bar{\eta}$ maps everything to zero, and

$$\bar{\Theta} \stackrel{def}{=} \left(\begin{array}{ll} & \{P \to P^{[i]} \mid P \in \Pi, i \in \mathcal{N}, 0 \le i < M_A\} \\ \cup & \{P \to P^{[M_A^{\ll}]} \mid P \in \Pi\} \\ \cup & \{P^{[c]} \to (c) \mid P \to (c) \in \Theta\} \\ \cup & \{P^{[M_A^{\ll}]} \to (c) \mid P \to (c) \in \Theta, c \ge M_A\} \\ \cup & \{P^{[i]} \to Q^{[j]}R^{[k]} \mid P \to QR \in \Theta, i = j +^{\langle M_A \rangle} k\} \\ \cup & \{P^{[i]} \to \min(Q^{[i]}, R^{[k]}) \mid P \to \min(Q, R) \in \Theta, i \le k\} \\ \cup & \{P^{[i]} \to \min(Q^{[k]}, R^{[i]}) \mid P \to \min(Q, R) \in \Theta, i \le k\} \\ \cup & \{P^{[i]} \to Q^{[i]} \mid P \to \max(Q, R) \in \Theta, i \ge \min[[R]]_A\} \\ \cup & \{P^{[i]} \to R^{[i]} \mid P \to \max(Q, R) \in \Theta, i \ge \min[[Q]]_A\} \end{array} \right)$$

Lemma 3 *Given an CAN A, $[[A]] = [[\bar{A}]]$ with $\bar{A}$ defined as in the above.*

Proof. ($\Longrightarrow$) We want to show by induction that for every $P \in \Pi$ and $t \in \mathcal{N}$, if $t \in [[P]]_A$, then $t \in [[P]]_{\bar{A}}$. In the base case, we have an execution like $P \to (c)$ in A. If $c \ge M_A$, then we have $P \to P^{[M_A^{\ll}]}$ and $P^{[M_A^{\ll}]} \to (c)$ in $\bar{\Theta}$. Otherwise, we have $P \to P^{[c]}$ and $P_c \to (c)$ in $\bar{\Theta}$. Thus the base case is proven.

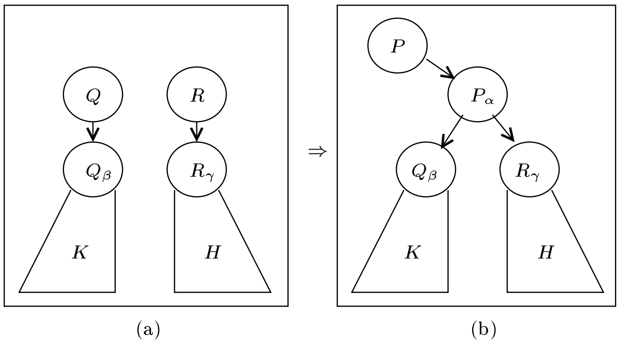


Fig. 3. Tree constructions for $\bar{A}$

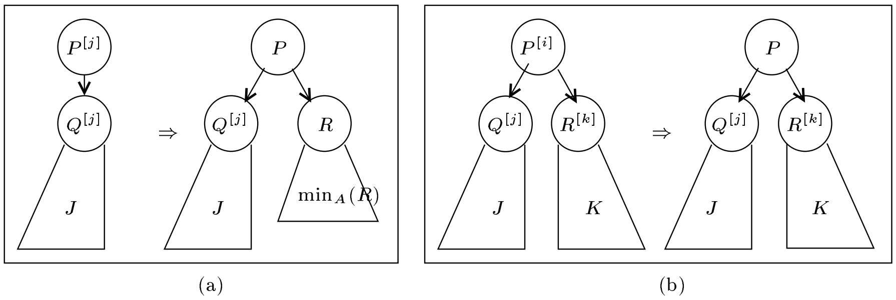


Fig. 4. Tree structure transformations

Now assume that this direction of the lemma is true for all execution trees in A of height $\leq h$. Given an execution tree in A of height $h+1$, there are two cases to consider. First, suppose that the root is constructed from a sequential rule $P \to QR$, and that the two subtrees have execution times J and K, respectively. According to the inductive hypothesis, we know that there are two execution trees in $\bar{A}$ shown as in figure 3(a) for some $j, k \in \{0, \ldots, M_A - 1, M_A^{\ll}\}$. Note that $M_A^{\ll}$ is actually a flag which means any value no less than M_A. Thus the values of j and k (or any execution time) must be either in $[0, M_A - 1]$ or marked as $M_A^{\ll}$. When $J \geq M_A$, $j = M_A^{\ll}$; otherwise, $j = J$. Similarly when $K \geq M_A$, $k = M_A^{\ll}$; otherwise, $k = K$. From these two trees, we can construct the tree shown in figure 3(b) for some $i = j +^{\langle M_A \rangle} k$ and prove the case. The case where the root corresponds to a max- or a min-rule can be proven similarly.

($\Longleftarrow$) We want to show by induction that for every $P \in \Pi$ and $t \in \mathcal{N}$, if $t \in [[P]]_{\bar{A}}$, then $t \in [[P]]_A$. Note that in $\bar{A}$, process types without superscripts, like P, only appear on the left-hand-sides of the rules. Assume that the tree in $\bar{A}$ for P with execution time t is T. The construction proceeds 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 4. Figure 4(a) is for max-rules like $P \to \max(Q, R)$ while figure 4(b) is for rules like either $P \to QR$ or $P \to \min(P, Q)$. Note that in figure 4(a), we use the drawing of a triangle with a circled R at the top and with $\min[[R]]_A$ inside the triangle as the execution tree

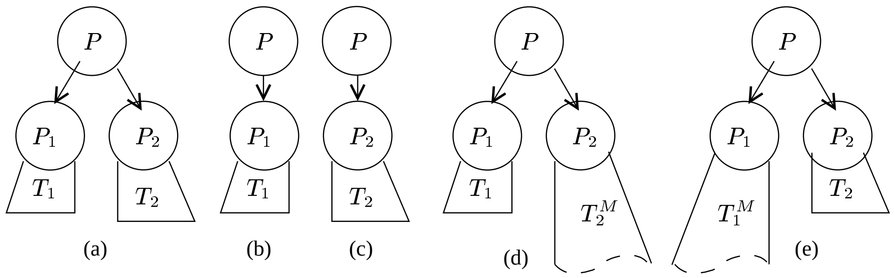


Fig. 5. Illustration of how to eliminate min-rules

of R with the minimum execution time for R. We perform this transformation for each original node (labeled with process types with superscripts) in the original execution tree T. The result after transformation is an execution tree for A and can be shown by induction on the structure with the same execution time as T. $\square$

4.3. Removing min*-rules*

Following lemma 3, from now on, we shall assume without loss of generality that all given CANs have neither arguments nor max-rules. As in the last section, we first give an intuitive explanation of our reduction from CANs with min-rules to CANs without. Then we present formal definitions and lemmas with proofs.

The idea of eliminating min-rules is very much similar to the idea of eliminating max-rules proposed in the last section. Given a min-rule like $P \to \min(P_1, P_2)$, we want to find out in what situation, the rule can be replaced by the following two rules: $P \to P_1$ and $P \to P_2$. Or equivalently, we want to find out in what situation, rule $P \to \min(P_1, P_2)$ helps generate the same set of execution times as rules $P \to P_1$ and $P \to P_2$ do. Also note that the set of execution times that can be constructed with rule $P \to \min(P_1, P_2)$ is always contained in the set of execution times that can be constructed with the two replacements. The reason can be seen from figure 5. Consider the execution tree in figure 5(a). Before replacement, the execution time of only one child subtree in figure 5(a) will definitely be included in the semantics of P. But after replacement, we have the two execution trees in figure 5(b) and (c). Therefore, the execution times of both child subtrees in figure 5(a) will be included in the semantics of P.

Thus the application of the replacement operation depends on the following question: *"Under what condition, does the set of execution times that can be constructed with rule $P \to \min(P_1, P_2)$ always contain the set of execution times that can be constructed with the two replacements ?"* One sufficient condition for this replacement operation is that

- no matter how large $[[T_1]]$ is, we can always find an execution tree T_2^M for P_2 such that $[[T_1]]$ is no greater than $[[T_2^M]]$; and
- no matter how large $[[T_2]]$ is, we can always find an execution tree T_1^M for P_1 such that $[[T_2]]$ is no greater than $[[T_1^M]]$ either.

In this situation, the execution tree shown in figure 5(b) is equivalent to the one shown in figure 5(d) as far as execution time is concerned. (The same is true for figure 5(c) to figure 5(e).) But what happens when $[[P_1]]$ or $[[P_2]]$ are of infinite sizes ? This situation can be broken down to the following four cases.

- If both $|[[P_1]]| = \infty$ and $|[[P_2]]| = \infty$, then the above-mentioned sufficient condition is naturally satisfied.
- Suppose $|[[P_1]]| = \infty$ while $|[[P_2]]| \neq \infty$. Then there is no restriction on propagating the execution times of P_2 to those of P. However, the execution times of P_1 that can be propagated using this rule to those of P are bounded by the maximum execution time of P_2.
- The case in which $|[[P_1]]| \neq \infty$ while $|[[P_2]]| = \infty$ is similar to the last case.
- Suppose both $|[[P_1]]| \neq \infty$ and $|[[P_2]]| \neq \infty$. Then the execution times that can be propagated using this rule to those of P are bounded by the smaller of the maximum execution times of P_1 and P_2.

According to analysis of the four cases, we find that it is important to know

- if a process type can generate an infinite number of execution times, and
- how long the maximum execution time of a process is if it is finite.

The following four sections are organized as follows. First, in section 4.3.1, we define and prove the necessary and sufficient condition for a process type to have infinitely many execution times. In section 4.3.2, we present and analyze the complexity of a bottom-up procedure used to decide which process types have infinitely many execution times. In section 4.3.3, we present an algorithm for calculating the finite maximums of process execution times. Finally in section 4.3.4, we wrap everything up and transform CAN into BCAN.

4.3.1. Finite Sizes of CAN Execution Time Sets

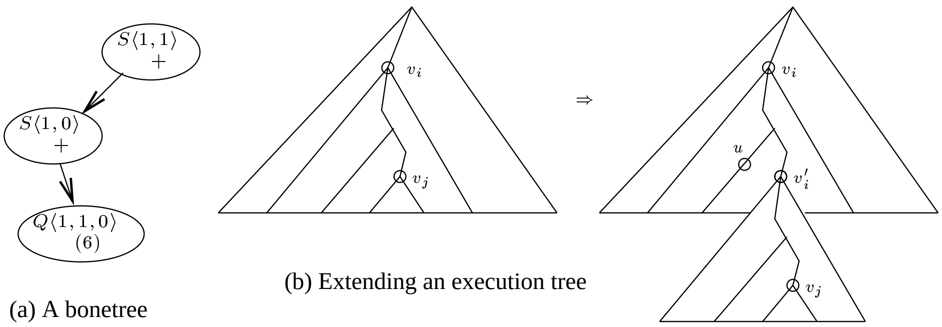
We define the following structures in an execution tree.

Definition 3 Bone trees *Given an execution tree $T = \langle V, E, r, \mu, \nu \rangle$ for $P \in \Pi$ in $A = \langle \Pi, \eta, P_0, \Theta \rangle$, a* bone tree *$B = \langle \bar{V}, \bar{E}, r, \mu, \nu \rangle$ of T is a substructure in T satisfying the following four conditons: (1) $\bar{V} \subseteq V$; (2) $\bar{E} = \{(v, v') \mid (v, v') \in E; v, v' \in \bar{V}\}$; (3) if $v \in \bar{V}$ and $\mu(v)$ is a rule like $P \to QR$, then exactly one of v's children is in $\bar{V}$; (4) if $v \in \bar{V}$ and $\mu(v)$ is a rule like $P \to \min(Q, R)$, then v's two children are both in $\bar{V}$.* □

The following condition of a bone tree is sufficient and necessary for the execution time set of a process type to be of infinite size.

Definition 4 Unboundedness condition *Given $P \in \Pi$ in $A = \langle \Pi, \eta, P_0, \Theta \rangle$, we say that 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 $i < j$, with the following two restrictions. (1) v_i and v_j are labeled with rules with the same left-hand-side. (2) 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.* □

For convenience of discussion, we use the term *repetition segment* to refer to the path segment from v_i to v_j along a bone tree path. v_i, v_j, and v_k are, respectively,



(a) A bonetree

(b) Extending an execution tree

Fig. 6. Bonetree and extending an execution tree

called the *starting*, *stopping*, and *significant* nodes along the repetition segment. A child node of v_k not in the bone tree is called a *pendant* of the bone tree.

We first want to establish the completeness and soundness of the unboundedness condition using lemma 4.

Lemma 4 *Given a CAN* $A = \langle \Pi, \eta, P_0, \Theta \rangle$ *and* $P \in \Pi$, $|[[P]]_A| = \infty$ *iff* P *satisfies the unboundedness condition.*

Proof. ($\Longrightarrow$) Let L_A be the largest constant used in definition A. Since $|[[P]]_A|$ is infinite, there must be an execution tree T for P heavier than $2^{|\Pi|}L_A$. Now we shall constructively identify a bone tree B in T which satisfies the unboundedness condition. Construction is accomplished by procedure bone() shown below.

---

bone(T) /* The root, left, and right subtrees are r, T_1, and T_2 respectively. */ {
  if r is a leaf, return T;
  else if r is labeled with an addition rule,
    if $[[T_1]] \geq [[T_2]]$, return tree(r, bone(T_1)); else return tree(r, bone(T_2));
  else if r is labeled with a min-rule,
    return tree(r, bone(T_1), bone(T_2));
}

For example, both of the execution trees shown in figure 1 generate the same bone tree depicted in figure 6(a) through procedure bone(). Now we want to show that B indeed satisfies the unboundedness condition.

We say a node is *heavy* if it is labeled with an addition rule, and if both of its subtrees have nonzero execution times. Other nodes are said to be *light*. We want to prove that along each path in B, there are at least $|\Pi| + 1$ heavy nodes. Note that the execution time of a subtree T' is divided into two nonzero execution times 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 execution time of the subtree is reduced to a value no less than half of the execution time of its parent tree. Also, if we branch through a min-node, the execution time of the child subtrees cannot be smaller than the execution time of its parent tree. Since $[[T]] > 2^{|\Pi|}L_A$, along any path of B, there will be at least $|\Pi|+1$ heavy nodes.

Finally, of the $|\Pi|+1$ heavy nodes along a path in B, two are labeled with rules having the same left-hand-side. Thus, this direction is proven.
($\Longleftarrow$) If there is such an execution tree T satisfying the unboundedness condition, we shall show that from T, we can construct another execution tree T' such that $[[T']] > [[T]]$ and T' satisfies the unboundedness conditon. Along every root-leaf path in B, we identify nodes v_i and v_j as described in the unboundedness condition. Note that the execution time of the subtree rooted at v_i is greater than the execution time of the subtree rooted at v_j due to the existence of a pendant along the repetition segment. Then we replace the subtree rooted at v_j with the one rooted at v_i. The replacement operation for one pair is shown in figure 6(b). We perform such a replacement operation for each root-leaf path in B. After replacement, the execution time of the tree rooted at v'_j is raised to the execution time of the tree originally rooted at v_i. Since the path contributes to the weight of the whole tree, it can be shown that the resulting new tree T' is heavier than $[[T]]$ and satisfies the unboundedness condition. □

4.3.2. Infiniteness of the Execution Time Set

With lemma 4, we can now develop an algorithm to compute the set of processes satisfying the unboundedness condition. The algorithm is embodied in the following bottom-up procedure, `unbounded()`.

`unbounded`(A) /* $A = \langle \Pi, \eta, S, \Theta \rangle$ */ {
  let $\Gamma = \{(P, \{(0, P)\}) \mid \min[[P]]_A \neq \infty\}$; (1)
  repeat until no more changes to Γ are possible, { (2)
    for each $P \to \min(P_1, P_2) \in \Theta$ and each $(P_1, \Lambda_1), (P_2, \Lambda_2) \in \Gamma$,
      add $(P, \{(c, P') \mid (c, P') \in \Lambda_1 \cup \Lambda_2; c = 0 \vee P' \neq P\})$ to Γ; (3)
    for each $P \to P_1 P_2 \in \Theta$ and each $(P_1, \Lambda_1) \in \Gamma$, ...(4)
      if $\min[[P_2]]_A = 0$, add $(P, \{(c, P') \mid (c, P') \in \Lambda_1; c = 0 \vee P' \neq P\})$ to Γ; (5)
      else if $0 < \min[[P_2]]_A \neq \infty$, add $(P, \{(1, P') \mid (c, P') \in \Lambda_1; P' \neq P\})$ to Γ; (6)
    for each $P \to P_1 P_2 \in \Theta$ and each $(P_2, \Lambda_2) \in \Gamma$, ...(7)
      if $\min[[P_1]]_A = 0$, add $(P, \{(c, P') \mid (c, P') \in \Lambda_2; c = 0 \vee P' \neq P\})$ to Γ; (8)
      else if $0 < \min[[P_1]]_A \neq \infty$, add $(P, \{(1, P') \mid (c, P') \in \Lambda_2; P' \neq P\})$ to Γ; (9)
  }
  return $\{P \mid (P, \emptyset) \in \Gamma\}$; (10)
}

The algorithm works on set Γ, whose elements are of the form (P, Λ), where Λ is, in turn, a set with elements like (c, P'). An element (P, Λ) represents an execution tree pattern with the following pattern:

- its root is labeled with a rule whose left-hand-side is P; and
- each element $(c, P') \in \Lambda$ represents an *obligation* to find a repetition segment, with a stopping node labeled with a rule whose left-hand-side is P', in a bottom-up manner. When $c = 0$, this means that P is still below the significant node (v_k) along the repetition segment. When $c = 1$, it means that

P is no lower than the significant node. An obligation is considered fulfilled when $P' = P$ and $c = 1$. A fulfilled obligation can be removed from Λ.

Note specifically that the obligations are represented as a set instead of as a multiset. That means that two obligations with the same pattern (c, P') can be fulfilled at the same time in a bottom-up manner when $P' = P$ and $c = 1$.

One important observation used in the definition of the unboundedness condition is that the exact execution time of a child tree rooted at a pendant is not important as long as we know that it is greater than 0. This observation affects our design of an element structure like (c, P') in Λ. Each obligation starts from the bottom at line (1) with $P' = P$ and $c = 0$ if $\min[[P]]_A \neq \infty$, i.e., there is an execution that begins at P. After initialization at statement (1), we iterate through the cycle for each combination of child subtree patterns when creating an execution tree pattern. This loop at statement (2) is very typical of bottom-up procedures. At the end of the h'th iteration, all execution trees of height $h+1$ have been considered.

At each iteration of the loop at statement (2), the execution tree pattern composition process is broken down into three cases. The first case is for min-rules, and obligations from both child subtrees are copied directly to the obligations of the parent execution tree unless we find that the obligation has been fulfilled.

The second and third cases are for sequential rules with bone repetition paths extending, respectively, from their left and right child subtrees to the parents. Accoding to the definition of bone trees, for nodes labeled with sequential rules, only one child will be included in the bone tree. The second case at statement (4) is for the case where the left child subtree is included in the bone tree, while the third case at statement (7) is for the case where the right child subtree is. Let us examine the second case at statement (4) in more detail. The existence of the left child subtree is justified by the existence of an element $(P_1, \Lambda_1) \in \Gamma$. The rule can generate a meaningful execution time only when $\min[[P_2]]_A \neq \infty$. Statement (5) tests whether the root of the new tree is not a significant node and copies the obligations of the child to those of the parent if the new root is not a significant node. Statement (6) tests whether the root of the new tree is a significant node, updates the obligations with $c = 1$, and records them with the new parent.

Note that at statements (3), (5), (6), (8), and (9), an obligation is passed to the parent only when it has not been fulfilled (i.e., $c = 0 \vee P' \neq P$).

Now we analyze the complexity of procedure `unbounded()`. First we want to know how complex Γ is. The elements of Λ are like (c, P'). There are $2|\Pi|$ possibilities. Each element in Γ is like (P, Λ). Thus, the number of possible elements in Γ is $|\Pi|2^{2|\Pi|} = |\Pi|4^{|\Pi|}$. For convenience, we let $H_\Gamma = |\Pi|2^{2|\Pi|} = |\Pi|4^{|\Pi|}$. This means that the loop at statement (2) can be executed at most H_Γ times since we only add elements to Γ and never remove one.

The complexity of each iteration of the loop at statement (2) is dominated by the first case, in which we consider every pair of elements in Γ. There are H_Γ^2 such pairs. To scan Λ_1 and Λ_2, we need to make $|\Pi|$ comparisons. Thus, each iteration of the loop at statement (2) has complexity $O(H_\Gamma^2|\Pi|)$. Thus, the complexity of the loop at statement (2) is $O(H_\Gamma H_\Gamma^2|\Pi|) = O(|\Pi|^4 16^{|\Pi|})$. This is the complexity of the whole algorithm since statements (1) and (10) do not cost much.

4.3.3. Finite Maximum of Process Execution Times

Given a process type P in CAN A, we let $\max[[P]]_A$ be the maximum execution time of process P in A. Conveniently, if $|[[P]]_A| = \infty$, we let $\max[[P]]_A = \infty$.

With the algorithm given in the last section, we can now decide whether $\max[[P]]_A = \infty$ for a given P. With this knowledge, we can further replace rules like $P \to \min(P_1, P_2)$ with $P \to P_1$, $P \to P_2$ if $\max[[P_1]]_A = \infty$ and $\max[[P_2]]_A = \infty$.

Now the remaining issue is how to compute the $\max[[P]]_A$ of all P with $\max[[P]]_A \neq \infty$. This can be done with the following procedure, `bound`(A).

`bound`(A) /* $A = \langle \Pi, \eta, S, \Theta \rangle$ */ {
    for each $P \in \Pi$, if $\max[[P]]_A = \infty$, then $M_P = \infty$, else $M_P = -\infty$;
    loop for $2^{|\Pi|} L_A |\Pi| + 1$ times, {
        For each rule $P \to (c) \in \Theta$, $M_P := \max(M_P, c)$;
        For each rule $P \to QR \in \Theta$, $M_P := \max(M_P, M_Q + M_R)$;
        For each rule $P \to \min(Q, R) \in \Theta$, $M_P := \max(M_P, \min(M_Q, M_R))$;
    }
}

It is seen from the proof of lemma 4 that for a P with $\max[[P]]_A \neq \infty$, $\max[[P]]_A \leq 2^{|\Pi|} L_A$. (Remember that in the first line of the proof for lemma 4, L_A is the notation for the biggest constant used in A.) Thus, within $2^{|\Pi|} L_A |\Pi| + 1$ iterations, a fixed point will be reached, and the maximum execution times of all processes can be found. At the end of the execution of the procedure `bound`(A), for each $P \in \Pi$ with $\max[[P]]_A \neq \infty$, $\max[[P]]_A = M_P$.

4.3.4. Semilinear Expressions of CAN

As explained at the beginning of section 4.3, the idea for eliminating min-rules follows the same paradigm given in section 4.2. Here we use symbol N_A for the time bound $2^{|\Pi|} L_A$ used in the proof of lemma 4. N_A plays the role of M_A described in section 4.2. Intuitively, for execution times beyond N_A, if any, the rules behave as there is no min-rules.

1. For each process type P, we replace it with the expanded process types $P^{[0]}, P^{[1]}, \ldots, P^{[N_A - 1]}, P^{[N_A^{\ll}]}$, where for each $0 \leq i < N_A$, $P^{[i]}$ means process type P with execution time exactly equal to i; and $P^{[N_A^{\ll}]}$ represents the process type P with execution time no less than N_A.

2. For each rule like $P \to \min(Q, R)$, we replace it with enumeration rules like

   $$\begin{array}{ll} P^{[i]} \to Q^{[i]}, & \forall (\max[[R]]_A \neq \infty \wedge i \leq \max[[R]]_A) \vee \max[[R]]_A = \infty; \\ P^{[N_A^{\ll}]} \to Q^{[N_A^{\ll}]}, & \forall \max[[R]]_A = \infty \end{array}$$

   Rules like $P^{[i]} \to Q^{[i]}$, with $i > \max[[R]]_A$, are not included since they do not generate an execution time that can be constructed with $P \to \min(Q, R)$. An enumeration rule like $P^{[i]} \to Q^{[i]}$ generates an execution time in the original semantics when $i \leq \max[[R]]_A$. Intuitively, the replacement operation enumerates all possible composition patterns with execution times less than N_A. When the execution time is no less than N_A, the composition pattern behaves as if no min-rules exist. Of course, the replacement operation can be done only if there exist execution trees of Q and R.

3. For the other rule types, we enumerate all composition patterns with expanded process types.

We can now reduce a CAN with no arguments and no max-rules to BCAN $\hat{A} = \langle \hat{\Pi}, \eta, P_0, \hat{\Theta} \rangle$ with $\hat{\Pi} = \{P_0\} \cup \{P^{[i]} \mid P \in \Pi, i \in \{0, \ldots, N_A - 1, N_A^{\ll}\}\}$ and

$$\hat{\Theta} \stackrel{def}{=} \left(\begin{array}{ll} & \{P_0 \to P_0^{[i]} \mid 0 \leq i < N_A; \max[[P_0]]_A \neq \infty\} \\ \cup & \{P_0 \to P_0^{[N_A^{\ll}]} \mid \max[[P_0]]_A = \infty\} \\ \cup & \{P^{[c]} \to (c) \mid P \to (c); \in \Theta; c < N_A; \} \\ \cup & \{P^{[N_A^{\ll}]} \to (c) \mid P \to (c); \in \Theta; c \geq N_A\} \\ \cup & \{P^{[i]} \to Q^{[j]} R^{[k]} \mid P \to QR \in \Theta; i = j +^{\langle N_A \rangle} k\} \\ \cup & \{P^{[i]} \to Q^{[i]} \mid P \to \min(Q, R) \in \Theta; i \leq \max[[R]]_A\} \\ \cup & \{P^{[i]} \to R^{[i]} \mid P \to \min(Q, R) \in \Theta; i \leq \max[[Q]]_A\} \end{array} \right)$$

Theorem 1 *Suppose we have a CAN $A = \langle \Pi, \eta, S, \Theta \rangle$, without arguments and without max-rules, and a BCAN $\hat{A}$ constructed according to the just-mentioned reduction. Then $[[\hat{A}]] = [[A]]$.*

Proof. True according to the construction of $\hat{A}$ and the results given in the last three subsubsections. □

5. CAN with Divergence Semantics

In real-world system designs, we usually have to make sure disasters do not happen even when some process fails to execute. We can also modify CAN theory to accommodate this possibility. We adopt *divergence semantics* such that when a process fails to execute, it never terminates, or equivalently, its execution time is ∞. This extension to CAN theory can be made quite naturally by adding the rule $P\langle X \rangle \to \infty$, for all $P \in \Pi$ and the argument variable sequence X, to component Θ in a CAN. The rule $P\langle X \rangle \to \infty$ means that process invocation $P\langle X \rangle$ never terminates and, thus, fails. Laws for arithmetic operations involving integers and ∞ also need to be naturally defined. Specifically, for all $c, d \in \mathcal{N} \cup \{\infty\}$, we want:

- $\infty + c = d + \infty = \infty + \infty = \infty$. The intuitive interpretation is that the sequential concatenation of a process execution with another failed process execution results in a failed process execution.
- $\max(\infty, c) = \max(d, \infty) = \max(\infty, \infty) = \infty$. Intuitively, this means that the parallel execution represented by max() fails if one of its child processes fails.
- $\min(\infty, c) = \min(c, \infty) = c$. Intuitively, this means that the parallel execution represented by min() succeeds if one of its child processes succeeds.

With the above divergence semantics, the transformation from general CAN to BCAN discussed in section 4 can still be applied with a minor modification to take care of additions and comparisons between integers and ∞. More interestingly, the transformation to CAN without min-rules can be made much simpler and more efficient. Since $\infty \in [[P_1]]_A$ and $\infty \in [[P_2]]_A$ now for all P_1, P_2, $P \to \min(P_1, P_2)$ can be unconditionally replaced by the two rules $P \to P_1$ and $P \to P_2$.

With the above divergence semantics, the BCAN transformed from $A = \langle \Pi, \eta, P_0, \Theta \rangle$ by eliminating all min-rules is $\dot{A} = \langle \Pi, \eta, P_0, \dot{\Theta} \rangle$ with

$$\dot{\Theta} \stackrel{def}{=} \left(\begin{array}{ll} & \{\theta \mid \theta \in \Theta; \theta \text{ is not a } \min\text{-rule.}\} \\ \cup & \{P \to P_1, P \to P_2 \mid P \to \min(P_1, P_2) \in \Theta\} \\ \cup & \{P \to \infty \mid P \in \Pi\} \end{array} \right)$$

Lemma 5 *Suppose we have a CAN $A = \langle \Pi, \eta, S, \Theta \rangle$ with divergence semantics, without arguments and without max-rules, and a BCAN $\dot{A}$ constructed according to the just-mentioned reduction. Then $[[\dot{A}]] = [[A]]$.*

6. Parallel CAN (PCAN) with a Rendezvous

We can also adopt a rendezvous semantics for the parallel compositions in CAN. That is, we can design a parallel rule like $P\langle X \rangle \to P_1\langle X_1 \rangle \| P_2\langle X_2 \rangle$ such that process $P\langle X \rangle$ has an execution tree if $P_1\langle X_1 \rangle$ and $P_2\langle X_2 \rangle$ have executions with the same execution time. With this alternative semantics, we can define Parallel CAN (PCAN) with the following syntax rules:

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

Here c_1, c_2 are natural numbers. The semantics of PCAN can also be defined using execution trees. Not all execution trees are legal because an internal node labeled with a parallel rule may have two subtrees with unequal execution times. 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, and then doing 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 is fulfilled by the simultaneous fulfillment of P_1 and P_2, respectively.

We shall only give a sketch of the semilinear expression construction of the semantics of PCAN. Details about the construction can be found in [50]. The execution of a PCAN may be viewed as consisting of repetitive spawning child processes. The execution state of a PCAN can be viewed as a set whose elements are like (t, P) such that the size of the set represents the number of distinct patterns of concurrent processes, t means the execution time accumulated from the beginning of the root process invocation, and $P \in \Pi \cup \{\perp\}$ is the process to be invoked next. Here $\perp$ means no more processes are to be invoked, i.e., the fulfillment of a parallel branch. Such an execution state is called a *snapshot*. The initial snapshot is $\{(0, P_0)\}$. An *accepting snapshot* is $\{(t, \perp)\}$ for some $t \in \mathcal{N}$. Note that we define a snapshot as a set instead of as a multiset. This means that all the descendant processes of the same pattern (t, P) terminate with the same execution time.

Numerically, the execution time of $(c_1)P_1(c_2)$ is the same as that of tail procedure-call $(c_1 + c_2)P_1$. We can define the relation $\texttt{NE-next}(B, (t, P), d, \theta, B')$ from a snapshot B to a possible numerically equivalent next snapshot B' by executing rule θ on process P and incrementing d to the current accumulated execution time t as

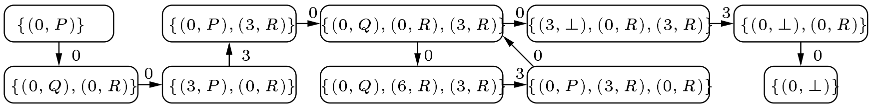


Fig. 7. subpiece of a normalized snapshot graph

shown below.

$\texttt{NE-next}(B, (t, P), d, \theta, B')$ /* Numerically equivalent next snapshot relation */ {
  if θ is $P \to (d)$,
    if $B' = (B - \{(t, P)\}) \cup \{(t + d, \bot)\}$, return *true*, else return *false*;
  else if θ is $P \to (c_1)Q(c_2)$,
    if $B' = (B - \{(t, P)\}) \cup \{(t + c_1 + c_2, Q)\}$, return *true*, else return *false*; . (1)
  else if θ is $P \to Q \| R$ and $d = 0$,
    if $B' = (B - \{(t, P)\}) \cup \{(t, Q), (t, R)\}$, return *true*, else return *false*,
  else return *false*.
}

A snapshot $\{(t_1, P_1), \ldots, (t_n, P_n)\}$ is *normalized* iff $(\exists 1 \leq i \leq n, t_i = 0) \wedge (\forall 1 \leq i \leq n, t_i \leq 2L_A)$. Given a snapshot $B = \{(t_1, P_1), \ldots, (t_n, P_n)\}$, we let $\texttt{normal}(B)$ be the normalized image of B, i.e., $\texttt{normal}(B) = \{(t - \min_{1 \leq i \leq n} t_i, P) \mid (t, P) \in B\}$.

Definition 5 : *normalized snapshot graph* The normalized snapshot graph of a PCAN is a directed graph (U, W, τ) such that U is the set of normalized snapshots, $W \subseteq U \times U$ is the set of arcs, and τ is a mapping from W to finite natural number sets. Formally speaking, given two $B, B' \in U$,

- $(B, B') \in U$ iff $\texttt{NE-next}(B, (t, P), d, \theta, \bar{B}) \wedge \texttt{normal}(\bar{B}) = B'$ for some $0 \leq t \leq 2L_A, P \in \Pi, 0 \leq d \leq 2L_A, \theta \in \Theta$, and snapshot $\bar{B}$;
- $\tau(B, B')$ is the set of time-progress amounts from B to B'. Precisely, $\tau(B, B') = \{d \mid 0 \leq d \leq 2L_A; \exists t \exists P \exists \bar{B}, \texttt{NE-next}(B, (t, P), d, \theta, \bar{B}) \wedge \texttt{normal}(\bar{B}) = B'\}$. □

Example 4 : *Figure 7 shows a subpiece of a normalized snapshot graph for a PCAN with six rules:* $P \to Q \| R; Q \to P(3); R \to (2)R(4); Q \to (3); R \to (3); R \to (0)$. *On each arc, the amount of time-progress is labeled. The subgraph characterizes some rule compositions with execution times* $6 + 3h$ *with* $h \in \mathcal{N}$. □

Furthermore, the following lemma shows that normalized snapshot graphs are both sufficient and necessary for the analysis of PCAN execution time sets.

Lemma 6 *Given a PCAN* $A = \langle \Pi, \eta, S, \Theta \rangle$, *for any* $t \in \mathcal{N}$, $t \in [[P]]_A$ *iff there is a path from* $\{(0, P)\}$ *to* $\{(0, \bot)\}$ *in the normalized snapshot graph of* A *such that the summation of time-progress amounts, one from the set label* $\tau()$ *of each arc along the path, is* t.

Proof. Sketch: An execution time $t \in [[P]]_A$ iff we can iteratively apply $\texttt{NE-next}()$ to $\{(0, P)\}$ and finally end up with $\{(t, \bot)\}$. This sequence of rule-applications generates a snapshot sequence. For every such snapshot sequence, we can construct an equivalent snapshot sequence with the same execution time by always choosing the element (t', P') with the smallest t' in each iteration. Then the normalized snapshots along this equivalent sequence constitute the path for the proof. □

The following lemma further shows that the normalized snapshot graph of any given PCAN is of finite size.

Lemma 7 *Given a PCAN* $A = \langle \Pi, \eta, S, \Theta \rangle$*, its normalized snapshot graph has at most* $2^{(|\Pi|+1)(2L_A+1)}$ *nodes.*

Then we can use the PCAN_time() procedure, in the following, to calculate the semilinear expressions of $[[P]]_A$.

---

set of integers $\quad S_{UU}$;
`PCAN_time`(A, P) {
  construct the normalized snapshot graph (U, W, τ) of A
  for each $(B_1, B_2) \in W$, $S_{B_1 B_2} := \tau(B_1, B_2)$;
  for each $(B_1, B_2) \notin W$ with $B_1 \neq B_2$, let $S_{B_1 B_2} := \emptyset$;
  for each $B_1 \in U$, let $S_{B_1 B_1} := S_{B_1 B_1} \cup \{0\}$;
  for each $B \in U$, for each $B_1, B_2 \in U$, let
    $S_{B_1 B_2} := S_{B_1 B_2} \cup \{i_1 + j i_2 + i_3 \mid i_1 \in S_{B_1 B}; i_2 \in S_{BB}; i_3 \in S_{BB_2}; j \in \mathcal{N}\}$; (1)
  return $S_{\{(0,P)\}\{(0,\perp)\}}$;
}

The algorithm is in the style of Kleene's closure algorithm. It is actually a modification of the bypass algorithms in [47, 48, 49]. Statement (1) means that we can go from B_1 to B_2 by first going from B_1 to B, then cycling through B any number of times, and finally going from B to B_2. The procedure iterates through all possible intermediate snapshots in the construction of a path from B_1 to B_2.

Theorem 2 *Given PCAN* $A = \langle \Pi, \eta, S, \Theta \rangle$ *and* $P \in \Pi$, $[[P]]_A =$ `PCAN_time`(A, P).

7. Stratified CAN (SCAN)

SCAN is designed to be a superclass of PCAN and CAN. It shows that we can mix the features of many CAN variations for the analysis of practical systems. In an SCAN A, conceptually, we have a sequence of PCANs and CANs $A_1 = \langle \Pi_1, \eta_1, P_{1,0}, \Theta_1 \rangle, \ldots A_n = \langle \Pi_n, \eta_n, P_{n,0}, \Theta_n \rangle$ such that

- for each $1 \leq i \leq n$, A_i is either a CAN or a PCAN;
- for any $1 \leq i < j \leq n$, if $\Pi_i \cap \Pi_j \neq \emptyset$, then $\Pi_i \cap \Pi_j = \{P_{j,0}\}$, and $P_{j,0}$ does not appear on the left-hand-side of any rules in Θ_i.

Formally speaking, SCAN $A = \langle \bigcup_{1 \leq i \leq n} \Pi_i, \bigcup_{1 \leq i \leq n} \eta_i, P_{1,0}, \bigcup_{1 \leq i \leq n} \Theta_i \rangle$.

Theorem 3 *SCAN is semilinear.*

Proof. The basic idea is that BCAN and semilinear expressions are equivalent. Details are in [50]. Thus we can evaluate from A_n to A_1 in sequence. Each time we get the semilinear expression of A_i, we translate it to a BCAN A'_i and replace A_i with A'_i. Eventually, the semilinear expression of A_1 will be constructed. □

8. Conclusion

Our major contribution is the construction algorithm of semilinear expressions of CAN systems. By viewing the min() and max() operations as parallel compositions in concurrent systems, CAN becomes an extension to classical context-free languages and process algebra with a new semantics. It will be interesting to see how the CAN theory can be used to enhance the analyzability of classic modeling languages, like timed automata, process algebra, WS1S, ..., etc.

Acknowledgements

The author would like to thank the reviewers of IJFCS for many helpful suggestions, which have improved the quality of the manuscript significantly. Actually, the observation after lemma 2 was restated excellently by one of the reviewers.

References

1. N. Audsley, A. Burns, M. Richardson, K. Tindell, A.J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. Software Engineering Journal, P.284-292, September 1993.
2. Alur, R., Courcoubetis, C., and Dill, D.L. (1993), Model-Checking in Dense Real-Time, *Information and Computation* **104**, Nr. 1, pp. 2–34.
3. R. Alur, T.A. Henzinger, P.-H. Ho. Automatic Symbolic Verification of Embedded Systems. in Proceedings of 1993 IEEE Real-Time System Symposium.
4. Alur, R., Henzinger, T.A., and Vardi, M.Y. (1993), Parametric Real-Time Reasoning, in "*Proceedings, 25th ACM STOC*," pp. 592–601.
5. J.C.M. Baeten, J.A. Bergstra, J.W. Klop. Decidability of Bisimulation Equivalence for Process Generating Context-Free Languages, Tech. Report. CS-R8632, 1987, CWI.
6. A. Boudet, H. Comon. Diophantine equations, Presburger arithmetics and finite automata. In Trees and algebra in programming - CAAP, LNCS 1059, 1995.
7. J.R. Burch, E.M. Clarke, K.L. McMillan, D.L.Dill, L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond, IEEE LICS, 1990.
8. A. Bouajjani, R.Echahed, P. Habermehl. On the verification Problem of Finite-State Concurrent Systems using Temporal Logic Specification, 10th IEEE LICS, 1995.
9. B. Berard, A. Labroue, P. Schnoebelen. Verifying performance equivalence for timed Basic Parallel Processes. 3rd FOSSACS, 2000, LNCS 1784, pp. 35-47.
10. R. Boyer, J.S. Moore. A Computational Logic Handbook, Academic Press, 1988.
11. J. Baeten, C. Middelburg. Process algebra with timing: real time and discrete time. In J. Bergstra, A Ponse, S. Smolka, eds., Handbook of Process Algebra, Chapter 10, pp.627-684, 2001.
12. A. Bockmayr, V. Weisphenning. Solving numerical constraints. Handbook of Automated Reasoning, editors: A. Robinson, A. Voronkov, Vol. I, Chapter 12, pp.751-842, Elsevier Science, 2001.
13. Clarke, E. and Emerson, E.A. (1981), Design and Synthesis of Synchronization Skeletons using Branching-Time Temporal Logic, in "*Proceedings, Workshop on Logic of Programs*," LNCS 131, Springer-Verlag.
14. Clarke, E., Emerson, E.A., and Sistla, A.P. (1986), Automatic Verification of Finite-State Concurrent Systems using Temporal-Logic Specifications, *ACM Trans. Programming, Languages, and Systems*, **8**, Nr. 2, pp. 244–263.
15. N. Chomsky. On certain Formal Properties of Grammar. Information and Control, **2**:2, 137-167.

16. W. Cheney, D. Kincaid. Numerical Mathematics and Computing, 3rd Ed, Brooks/Cole Publishing, Pacific Grove, CA, 1994.

17. J.-Y. Choi, I. Lee, H.-L. Xie. The specification and schedulability analysis of real-time systems using ACSR. 16th IEEE RTSS, 1995.

18. G.E. Collins. Quantifier elimination for real closed fields by cynlindrical algebraic decomposition. Automata Theory and Formal Languages, editor: H. Brakhage, LNCS 33, pp.34-183, 1975.

19. J. Crow, S. Owre, J. Rushby, N. Shankar, M. Srivas. A tutorial introduction to PVS. Workshop on Industrial-Strength Formal Specification Techniques, Boca Raton, Florida, April 1995.

20. P. Downey. Undecidability of Presburger arithmetic with a single monadic predicate letter. Technical Report 18-72, Center for Research in Computing Technology, Harvard University, 192.

21. J. Engblom, A. Ermedahl, M. Sjoedin, J. Gubstafsson, H. Hansson. Worst-Case Execution Time Analysis for Embedded Real-Time Systems. Journal of Software Tools for Technology Transfer, 2001, 14.

22. K. Fukuda, A. Prodon. Double description method revisited. Combinatorics and Computer Science, LNCS 1120, pp.91-111, Springer-Verlag, 1996.

23. A. Gupta, A.L. Fisher. Parametric circuit representation using inductive boolean functions. CAV'93, LNCS 697, pp.15-28, 1993.

24. A. Gupta, A.L. Fisher. Representation and symbolic manipulation of linearly inductive Bolean functions. IEEE ICCAD'93, pp.192-199, IEEE Computer Society.

25. S. Ginsburg, S.A. Greibach. Deterministic Contex-Free Languages, Information and Control, **9**:6, 563-582.

26. J. Glenn, W. Gasarch. Implementing WS1S via finite automata. In Automata Implementation, WIA'96, LNCS 1260, 1997.

27. E.M. Gurari, O.H. Ibarra. The complexity of decision problems for finite-turn multicounter machines, Journal of Comput. and Syst. Sc. **22**, 1981, pp.220-229.

28. P.E. Gill, W. Murray, M.E. Wright. Numerical Linear Algebra and Optimization, Vol. 1., Addison-Wesley, 1991, Redwood City, California.

29. D. Harel. Statecharts: A visual Formalism for Complex systems. Science of Programming **8**, 1987.

30. J.G. Henriksen, J. Jensen, M. Jorgensen, N. Klarlund, B. Paige, T. Rauhe, A. Sandholm. MONA: Monadic second-order logic in practice. TACAS'95, LNCS 1019, 1996.

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

32. T.A. Henzinger, X. Nicollin, J. Sifakis, S. Yovine. Symbolic Model Checking for Real-Time Systems, IEEE LICS 1992.

33. C.A.R. Hoare. Communicating Sequential Processes, 1985.

34. O.H. Ibarra. Reversal-bounded multicounter machines and their decision problems, JACM **25**, 1978, pp.116-133.

35. O.H. Ibarra, T. Bultan, J. Su. On reachability and safety in infinite-state systems. International Journal of Foundations of Computer Science, Vol. 12, Nr. 6, 2001, pp.821-836, World Scientific Pub.

36. F. Jahanian and A.K. Mok. Safety analysis of timing properties in real-time systems, IEEE Trans. Software Engineering, Vol.SE-12, No9, 1986, pp. 890-904.

37. Farnam Jahanian, A.K. Mok. Modechart: A Specification Language for Real-Time systems, IEEE Transactions on Software Engineering, vol. 20, no. 12, Dec. 1994.

38. P. Kelb, T. Margaria, M. Mendler, C. Gsottberger. Mosel: a flexible toolset for monadic second-order logic. CAV'97, LNCS 1217, 1997.

39. S. Lasota. Decidability of strong bisimiliarity for timed BPP. CONCUR'2002, LNCS 2421, Springer-Verlag.

40. B. Murtagh. Advanced Linear Programming. McGraw Hill, 1981.

41. M. J. Meyer, H. Wong-Toi. Schedulability Analysis of Acyclic Processes. IEEE RTSS'98.

42. T. Nipkow, L.C. Paulson, M. Wenzel. Isabelle/HOL: A proof assistant for Higher-Order Logic. LNCS 2283, Springer-Verlag.

43. 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.

44. R.J. Parikh. On Context-Free Languages, JACM **4**, 1966, 570-581.

45. W. Pugh. Counting solutions to Presburger formulas: how and why. ACM SIGPLAN PLDI, 1994.

46. J.E. Vuillemin. On circuits and numbers. IEEE Transactions on Computers 43(8):868-879, 1994.

47. F. Wang. Scalable Compositional Reachability Analysis of Real-Time Concurrent Systems. In Proceedings of the 2nd IEEE RTAS, Boston, June, 1996.

48. F. Wang. Parametric Timing Analysis for Real-Time Systems, Information and Computation, Vol. 130, Nr 2, Nov. 1996, Academic Press, ISSN 0890-5401; pp 131-150. Also in "*Proceedings, 10th IEEE Symposium on Logic in Computer Science*," 1995.

49. F. Wang. Parametric Analysis of Computer Systems, *Formal Methods in System Design*, pp.39-60, 17, 39-60, 2000. Also in *AMAST'97*, LNCS 1349 (w. P.-A. Hsiung)

50. F. Wang. Inductive Composition of Numbers with Maximum, Minimum, and Addition - A New Theory for Program Execution-Time Analysis (Full version). ACM repository.

51. P. Wolper, B. Boigelet. An automata-theoretic approach to Presburger arithmetic constraints. SAS'95, LNCS 983, pp.21-32, Springer-Verlag.

52. F. Wang, A. Mok. RTL and Refutation by Positive Cycles, in Proceedings of the Formal Methods Europe Symposium, Barcelona, Spain, October 1994, LNCS 873.

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

54. F. Wang, H.-C. Yen. Parametric Optimization of Open Real-Time Systems, proceedings of SAS 2001, Paris, July 2001, LNCS 2126, Springer-Verlag.

55. F. Wang, H.-C. Yen. Timing Parameter Characterization of Real-Time Systems. 8th CIAA, LNCS 2759, Springer-Verlag; July 16-18, 2003.

56. G. Xie, C. Li, Z. Dang. New complexity results for some linear counting problems using minimal solutions to linear diophantine equations. 8th CIAA, 2003, LNCS 2759, Springer-Verlag.