Applications of Abstract Algbera in Computational Concurrency

Arisa Cowe

April 2022

1 Introduction

Within the realm of computer science, concurrency is a concept that comes up quite often and for good reason: It is necessary for the development of efficient computing systems, whether that be large scale networks computers or advanced machine learning softwares, as concurrency focuses on the most effective use of limited shared resources across computing components, such as data, computer memory, network connections, available computer cores, etc. In general, concurrency in computing science refers to the ability of a program to execute its tasks (processes) in partial, overlapping order; That is, the ability for a program to deal with and complete many of its tasks at once by executing portions of them. Thus, concurrent computing is computation where several instructions (for a process) are executed within overlapping time periods rather than having processes being executed sequentially, or one-at-a-time. The benefit of executing processes in this manner is that the waiting time from asymmetric resource use is significantly reduced, thus making the program more efficient than if it were executed sequentially. This leads into concepts such as threading, the division of such processes into portions (threads) that can be executed concurrently such that the resources usage is maximized and wait time (deadlocks) are minimized. Of course a fundamental question that comes up in threading is "How can our program be partitioned and arranged to best utilize the scarce resources we have?", the answer to which requires understanding of process behaviors and interactions as well as the resulting outcomes of such. One such way to solve this problem is with process algebra. Process algebras, also known as process calculi, are mathematical approaches to modeling concurrent systems. Since the early 1970's, there have existed many process algebras including but not limited Communicating sequential processes (CSP), Calculus of communicating systems (CCS), Algebra of communicating processes (ACP), Language Of Temporal Ordering Specification (LOTOS), and π-calculus, all of which help its users model, reason, and understand computational processes and their interactions. The process algebras that will be explored in this paper are ACP and thread algebra.

2 The Algebra of Communicating Processes

The algebra of communicating processes, or ACP, is a process algebra developed by Jan Bergstra and Jan Willem Klop of the University of Amsterdam in 1982. Within this algebra there exists a multitude of operations that are closed over a set of non-empty processes. Such processes are composed of one or more atomic actions, or indivisible, uninterruptible actions (i.e. a computer instruction). We begin with Bergstra and Klop's basic process algebra (BPA) which only consists of the binary alternative composition operator, + , and sequential composition operator, $\cdot$ over the set of processes. That is, given processes X and Y, $X + Y$ denotes the process that either executes X or Y and $X \cdot Y$ denotes the process that executes X, then Y Here are the axioms that define this BPA:

- $X + Y = Y + X$
- $(X + Y) + Z = X + (Y + Z)$
- $X + X = X$
- $(X + Y)Z = XZ + YZ$
- $(XY)Z = X(YZ)$

We observe that set of processes is associative and commutative under + and is right distributive under $\cdot$. However, there does not exist an identity process, as there is no "empty" process defined in the set, nor an inverse process for any process, as these processes are considered irreversible in the running of the program. Thus, this BPA forms an abelian semigroup under + and *, a type of algebraic structure defined as a set under a binary operation where the operation is associative but there does not necessarily exist identity element or inverse elements. However, the set of processes, and in turn the BPA, can be extended to include the special action, δ (i.e. the deadlock action) where $\delta + X = X + \delta = X$and $\delta \cdot X = X \cdot \delta\delta$. In this extended BPA, BPA$_delta$,δ can be considered an "empty" process in alternative composition as it represents an action that is "waiting" for resources and thus cannot be executed. In turn, the other, non-deadlocked process takes precedence in this branching computation as that action can actually be executed. Thus, δ is the "additive" identity in BPA$_\delta$ which makes extended set of processes in BPA$_\delta$ an abelian monoid under +, a type of algebraic structure defined as a set under a binary operation where the operation is associative and there exists an identity element but not necessarily inverses for all processes. Though the BPA, and its extension BPA$_\delta$, are relatively simple algebras compared to its extensions, it along with its algebraic structures already provides a great deal of information about the execution of processes, executive precedence of computations (operations) over processes, and executive equivalence between syntactically different processes (ie. what processes are executively the same?) In addition, the current BPA$_\delta$ can be further extended with other binary operators to better model concurrency across processes. This includes the the merge operator, $||$, which represents two concurrent processes, the left merge operator $||_{\_}$ respectively, representing two concurrent processes whose leftmost action takes executive precedence over other actions, are and the communication merge operator —, which represents two processes directly interacting with each other via a specified communicator function, then merging (i.e., one process "sends", the other one "receives" at the same time). That is, for two processes $X = (ab)$ and $Y = (cd)$, $X||Y$ generates the action sequences $abcd, acbd, acdc, cabd, cadb, cdab$ while $X||\_Y$ can only generate the action sequences $abcd, acbd$, and $acdc$ due to a's executive precedence. Here are some the axioms of these operations:

- $X||Y = X||\_Y + Y||\_X + X|Y$
- $A|| = AX$
- $AX|| = A(X||Y)$
- $(X + Y)||\_Z = (X||\_Z) + (Y||\_Z)$
- $AX|B = A|BX = (A|B) \cdot X$
- $AX|BY = (A|B)(X||Y)$
- $(X + Y)|Z = X|Z + Y|Z$
- $X|(Y + Z) = X|Y + X|Z$

These merge operators form a magma, a set closed under a binary operator but with no other properties beside that, over the set of processes. Despite this, it is interesting to see that some of the merge operators, such as the left and communication merges, are right-distributive over alternative composition, demonstrating executive equivalence of merge operations over alternative processes with alternation of merged processes. This structure, coupled with personal observation of the axioms provides a great deal of information about the executive precedence and equivalences of concurrent processes. In particular the lack of properties merge operators themselves demonstrate the uniqueness of the permutations of actions; That is, for the results something like $a||b$, the results, ab and ba, are all unique processes. There are plenty more operators and symbols that can extend this algebra, such as the abstraction/silent operator τ and encapsulation, but these operations are more than enough to demonstrate the versatility of this algebra. Even with just these five operators and the action symbols including δ, we can already start to see how this can be used to identify patterns, prove executional truths, and solve problems within the realm of developing concurrent and/or parallel computing systems whether that be proving the mutual exclusivity of concurrent processes (i.e. the non-overlap of resource usage by processes) or verifying the correctness of computer-integrated manufacturing (CIM) system architectures (i.e. an automated manufacturing system works as intended). From here, we will move onto discussing thread algebra and its applications in computational concurrency and (thread-level) parallelism.

3 Thread Algebra

Thread algebra is an extension of ACP developed by Jan Bergstra alongside C.A. Middelburg. Unlike ACP however, this algebra is a polarized process algebra; That is, instead of working with processes whose actions are all functionally equivalent in the execution of the program, this process algebra involves processes whose actions can be categorized into distinct request and non-request type actions. The core of thread algebra, basic thread algebra (BTA), is one that operates on threads, through the processing of request actions by a service. In particular, there exists two constant threads D and S where D denotes a deadlocked (incomplete, waiting) thread and S denotes a terminated (completed) thread. There also exists a post conditional composition of a request action operator $\trianglelefteq a \trianglerighteq$ that operates over two threads, where a valid reply from the service on an action yields the first thread and an invalid reply yields the second thread. Finally, the action prefix a‘ is shorthand for the expression $P \trianglelefteq a \trianglerighteq P$. Given the only axiom of BTA is $x \trianglelefteq \tau \trianglerighteq y = x \trianglelefteq \tau \trianglerighteq x$, the structure formed can only be classified as a magma at best. However, this lack of rigidity allows for the mapping of processes from ACP or even programs from program algebra to constants in thread algebra via magma homomorphisms. This is greatly helpful in understanding what processes and programs result in what kind of threads allowing its users to devise arrangements of concurrent processes and programs that maximize the efficiency of the system. In addition, BTA can be extended to incorporate services and their interaction with a program or process, thread vectors, and special operators such as the “use” or “poly thread” operators that allows its users to determine how how sequentially parallel processes and programs can be made into interleaved multithreaded programs, that is, programs that carry out tasks through the overlapping of programs with alternating independent sequences. Should the hardware allow for it (i.e. multicore processors) multithreaded programs can be broken down into programs, based on what thread they are in, to be executed simultaneously. (i.e. thread level parallelism).

4 Conclusion

While the algebraic structures explored in this paper stretch far beyond the realm of the groups, rings, and fields mentioned in MATH171, they still embody systems of logic over a collection of objects like those structures. That is, these algebraic structures can be viewed as extensions of the structures observed in class such as group-like structures which include semigroups and monoids. By having these structural points of reference, we can extend mathematical reasoning from elementary algebras onto number-like objects such as programs and in-program processes, whose intrinsic value is defined by what actions are executed (i.e, state). That is, by understanding the base structures, its users have a means of visualizing a system of elements and their interactions based on given structural constraints. From this, patterns within the elements can be identified and their interactions can be reasoned with in order to solve problems, especially problems involving much complexity as is the case with developing concurrent and parallel programs and systems.

Works Cited

Bergstra, J. A., and J. W. Klop. “An Introduction to Process Algebra.” *Applications of Process Algebra*, edited by J. C. M. Baeten, Cambridge University Press, Cambridge, 1990, pp. 1–22. Cambridge Tracts in Theoretical Computer Science.

Bergstra, J. A., and C. A. Middelburg. “Thread Algebra for Poly-Threading.” *Formal Aspects of Computing*, vol. 23, no. 4, 2011, pp. 567–583., doi:10.1007/s00165-011-0178-3.

Bergstra, J. A., and C. A. Middelburg. “Thread Algebra for Strategic Interleaving.” *Formal Aspects of Computing*, vol. 19, no. 4, 2007, pp. 445–474., doi:10.1007/s00165-007-0024-9.

Mauw, S. “Process Algebra as a Tool for the Specification and Verification of CIM-Architectures.” *Applications of Process Algebra*, edited by J. C. M. Baeten, Cambridge University Press, Cambridge, 1990, pp. 53–80. Cambridge Tracts in Theoretical Computer Science.

Nieuwland, Eric R. “Proving Mutual Exclusion with Process Algebra.” *Applications of Process Algebra*, edited by J. C. M. Baeten, Cambridge University Press, Cambridge, 1990, pp. 45–52. Cambridge Tracts in Theoretical Computer Science.

Ponse, A., van der Zwaag, M.B. (2006). “An Introduction to Program and Thread Algebra.” *Logical Approaches to Computational Barriers*, edited by A. Beckmann, U. Berger, B. Löwe, J.V. Tucker, Springer, Berlin, Heidelberg, 2006, pp. 459-472. Computability in Europe (CiE).