A Survey of Reasoning Methods for Concurrent Systems

Chun-Kun Wang
Department of Computer Science
Univ. of North Carolina at Chapel Hill
Chapel Hill, NC 27599, USA
Email: amos@cs.unc.edu

Hao Xu
Data Intensive Cyber Environment Center
Univ. of North Carolina at Chapel Hill
Chapel Hill, NC 27599, USA
Email: xuh@cs.unc.edu

***Abstract*—The theory of parallel and distributed systems in computer science has been developing for decades. Multiple methods have emerged for the purpose of providing flexibility, expressiveness and theories for concurrent processes. This paper provides a comparative introduction to the history of concurrent systems reasoning and a decision tree to guide the choice of a reasoning method accordingly. Our account on these matters is incomplete.**

***Index Terms*—Concurrent systems, Formal methods, Program logic, Process calculi.**

I. Introduction

Concurrency theory has been developing for decades, yet remains an unsolved problem today. Modern software heavily relies upon the services provided by distributed systems, ranging from smartphone applications to social media. It has been long recognized that fault-tolerance, efficient concurrency and performance are of crucial importance for concurrent systems.

Shared variables play an essential role in concurrency. Complex concurrency often exhibit unexpected failures: faults may occur arbitrarily in any component or at any point, and networks may experience packet losses, reordered packets and duplicate packets. These concurrency patterns are driving the need for further specification toward verifying the consistency and the correctness to meet strict trustworthiness requirements. Reasoning about shared state, however, has becomes a difficult problem, because concurrent threads simultaneously work with shared data.

Further, the implementation of critical section requires either coarse-grained synchronization (i.e., locks or semaphores) or fine-grained synchronization (i.e., compare-and-swap). Formal methods for concurrency reasoning need to provide a corresponding semantics capable of reasoning functional correctness. Also, in the design stage, abstraction and modularity are able to hide irrelevant details and ease the burden for programmers implementing a high-quality concurrent system that satisfies the requirements of correctness, resource-efficiency or fault-tolerance.

As discussed above, verification of programs by reasoning methods is the only known way to guarantee that a software is free of programming errors and meets the requirements or the features we need. For many decades, there have been substantial studies of formal methods for concurrency reasoning. In this paper, we do not compare reasoning methods down to the last detail (i.e., syntax, inference rules and proof systems). We do, however, discuss concurrency reasoning approaches from a high-level and provide a comparative introduction to them.

The authors would like to stress that our account on the developmental history of concurrent systems reasoning is incomplete. The guides to select a method accordingly are provided in this paper. Others may well have very different views.

II. History

The history of reasoning concurrency dates back to the late twentieth century. The foundations of early concurrency theory can be found in Petri nets, automata theory and formal languages. Today, concurrent systems reasoning can be separated into two families: one is program logics started from Hoare logic [1]; the other is process calculi deriving from Hoare's CSP [19] and Milner's CCS [27].

The family of program logics includes assertional reasoning [2], rely-guarantee reasoning [3], [4], separation logic [7] and concurrent separation logic [11], [12]. The family of process calculi had an early stage of development, including CSP, ACP [20] and CCS. Later, there were multiple descendants, such as LOTOS [22], π-calculus [27], ambient calculus [42] and join calculus [30]. The lineage tree of methods for reasoning concurrency is shown in Figure 1. It is organized roughly in chronological order to present the development of concurrency theories. Although session-types [31] and behavioral-types [41] belong to the family of process calculi, we consider them as a new family because they are built upon the theory of type systems.

III. Concurrency reasoning

The reasoning of concurrent systems has taken two major directions, as shown in Figure 2. The approach of process calculi is introduced when interactions and communications between a collection of independent agents or processes are focused. The approach of program logic is suited for reasoning entire programs when programming paradigms matter. Once the reasoning direction is decided, a proper method can be chosen, according to the discussion in the sections that

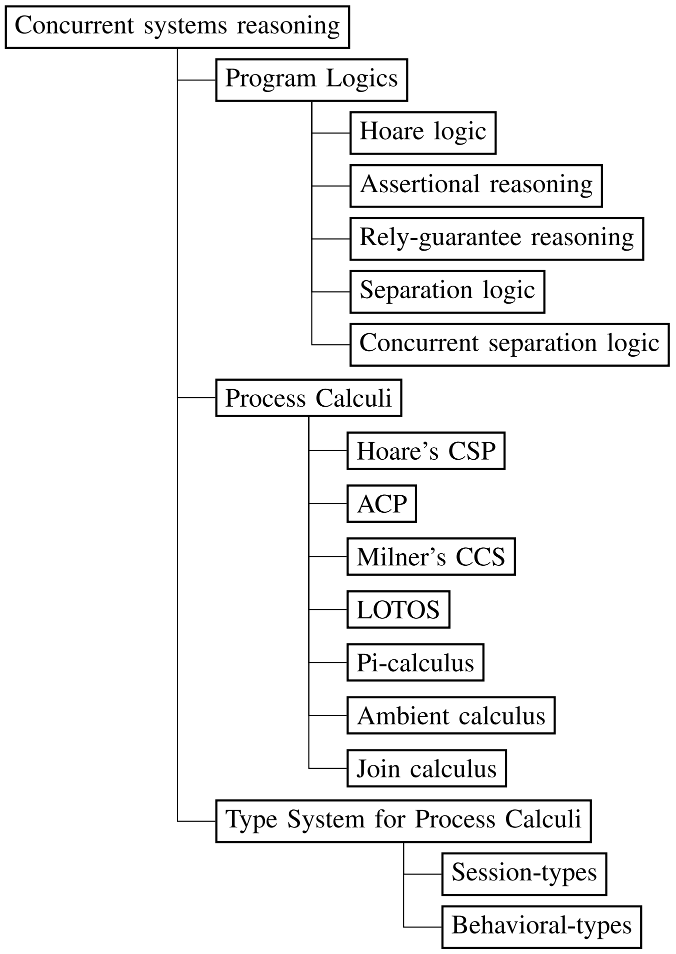

Fig. 1. Lineage tree of different categories and methods for concurrent systems reasoning, roughly organized chronologically.

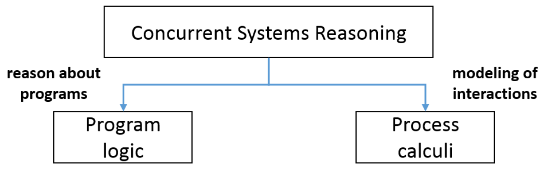

Fig. 2. Two major directions for reasoning of concurrent systems.

follow. The following section provides a broad overview of reasoning methods rather than a detailed review (i.e., syntax and inference rules).

A. Program Logic

Hoare logic is the first formal method for program logic. A programming paradigm is transformed into a *Hoare triple*. *Hoare triple* is of the form $\{P\}\ C\ \{Q\}$, in which P is the precondition, C is a command and Q is the postcondition. Precondition and postcondition are assertions in predicate logic. With the axioms and inference rules provided by Hoare logic, a sequentially imperative programming can be verified.

Based on the foundation of Hoare logic, assertional reasoning combines Hoare logic and linear-time temporal logic to reason shared variables in programs. It formalizes the access to a shared variable (i.e., a critical section) being presented as a simple state transition. The verification of assertional reasoning can then be done by using temporal logic proof systems.

Rely-guarantee method splits a specification of concurrent systems syntactically into two corresponding conditions, the rely and the guarantee. It gives a systematic way of describing the state changes performed by the environment or by the program, respectively. The rely-guarantee approach is also used to reason asynchronous programs [6] and dynamic distributed systems [40].

Although the rely-guarantee method does its job, its specifications are complex because the entire state is described in global interference specification. It increases the reasoning complexity when the program state updates. The interference specification must be checked against every state update. Separation logic solves the problem discussed above by offering a *separating conjunction* $*$. For instance, $P * Q$ asserts that P and Q are disjoint memory addresses. *Separating conjunction* makes specifications local and reduces the reasoning work. Vafeiadis et al. [5] balance the trade-offs between the rely-guarantee method and separation logic, because separation logic has its own difficulty with interference. Separation logic is widely used for shared data structures reasoning [8], [9], [10].

Concurrent separation logic (CSL) takes one more step to express concurrency. It provides a parallel composition to describe concurrent threads. For example, $P \parallel Q$ represents that thread P and thread Q are executed concurrently. CSL also preserves the private resources, which are invisible to the environments, for each thread. With these features, CSL is able to support thread modularity and memory modularity [13]. Many works extend CSL for modular reasoning to support fine-grained concurrency (i.e., compare-and-swap instructions) [14], [15], [16], [17], [45], [46].

Figure 3 presents a decision tree for choosing validation methods in the category of program logic. Hoare logic is a set of logical rules for reasoning sequential programs. As regards concurrent programs, assertional reasoning combines Hoare logic with temporal logic to reason shared variables. R/G reasoning splits specification into rely and guarantee parts, which provides systematic reasoning when the number of shared variables becomes large. Reasoning memory pointers requires more extensions (i.e., shape analysis and separating conjunction) provided by separation logic. Concurrent separation logic supports thread modularity and memory modularity, which allow each thread to have its private resources to reduce the work of reasoning.

B. Process Calculi

Process calculi are algebraic languages to describe interactions and communications between a collection of independent agents or processes. The early development of process calculi

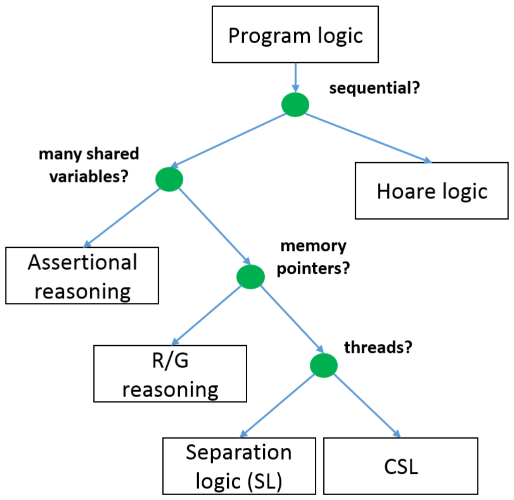

Fig. 3. Decision tree for choosing validation methods in the category of program logic. A circle represents a decision node, which is a condition rule for the feature of concurrent systems. The outcome of TRUE goes right node, while the outcome of FALSE goes left node.

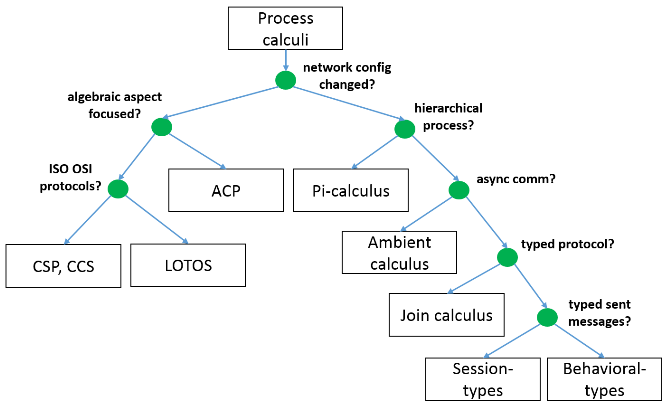

Fig. 4. Decision tree for choosing validation methods in the category of process calculi. A circle represents a decision node, which is a condition rule for the feature of concurrent systems. The outcome of TRUE goes right node, while the outcome of FALSE goes left node.

mainly focused on ACP, CSP and CCS. Many works [21], [24], [25], [47], [48] compare the difference between ACP, CSP and CCS. The development of CSP and CCS are similar because they influenced one another. For instance, their fundamental operations both have sequencing, nondeterministic choice and parallel composition. The main difference between CSP and CCS lies in their synchronization mechanisms. CCS's communication adopts a binary handshake via matching actions, which is combined with abstraction. CSP's communication uses multi-way synchronization combined with restriction. ACP emphasizes the algebraic aspect and has a more general communication scheme. LOTOS, combining CSP and CCS, was developed in response to the protocol specification in ISO OSI standards.

π-calculus was first introduced by Robin et al. [27]. It is an extension of the process algebra CCS and a type of process calculus designed for representing parallel computation. Since its introduction, it has been widely used in many applications. Abadi et al. [43] extended π-calculus to have more primitives for encryption and decryption. π-calculus has also been extended to Business Process Modeling Language (BPML) [44]. Another advantage of π-calculus is that it can handle dynamic network configuration by allowing channels to be passed as data along the channels themselves. π-calculus influences the way to specify distributed systems [28]. Pict [29] is a programming language based on π-calculus.

Unlike π-calculus, ambient calculus has the ability to model hierarchical process boundaries. It formalizes the distributed system components, including nodes, channels, messages, and mobile code. With these system components, ambient calculus is able to describe the situation where entire computational environments are changed, like dynamic network topology.

Join calculus is considered as an asynchronous π-calculus, which avoids rendezvous communications in other process calculi. The implementation of atomic non-local interaction is difficult in distributed settings. Join calculus can be encoded into π-calculus and vice versa.

Session-types has became a popular way to verify the protocols [32], [34], [49]. Session-types is considered to be a type of π-calculus that describes communication protocols, in which programs can be checked to see if they conform to protocols either statically (at compile-time) or dynamically (at runtime) [33]. Session-types reasoning [35] is built upon a Curry-Howard correspondence between session-types and the process model of π-calculus. This result has influenced much other research in classical linear logic [37], [38], [39] and also led to the birth of behavioral-types [36], [41], [50], [51]. Behavioral-types enriches the expressiveness of session-types to specify its expected patterns of interaction, rather than treat communication channels as raw data types (i.e., bytes arrays or strings). This feature preserves all the benefits and guarantees provided by type systems while verifying protocols.

A decision tree for choosing validation methods in the category of process calculi is shown in Figure 4. The first factor should be considered is whether network configuration may change during the computation. If network configuration is static, a basic process algebra method, like ACP, CSP, or CCS, can be adopted. LOTOS serves the purpose of ISO OSI protocol standards. Under the circumstance of changed network configuration, the chosen reasoning methods must be in the family of π-calculus. Join calculus is able to formalize asynchronous communication, such as no rendezvous communications. If the protocol is complex and large, the support from type systems is needed. Behavioral-types is selected when the sent or received data must be typed; otherwise, session-types will be chosen.

IV. Conclusion

Modern software heavily relies upon the services provided by distributed systems. It has been long recognized that fault-tolerance, efficient concurrency and performance are of crucial importance for concurrent systems. These features are driving the need for further specification toward verifying consistency and correctness. The theory of parallel and distributed systems in computer science has been developing for decades. This paper provides a comparative introduction to the history of concurrent systems reasoning and a decision tree to guide the choice of a reasoning method accordingly.

References

[1] Floyd, Robert W. *Assigning meanings to programs.* In Program Verification, pp. 65-81. Springer, Dordrecht, 1993.

[2] Shankar, A. Udaya. *An introduction to assertional reasoning for concurrent systems.* ACM Computing Surveys (CSUR) 25, no. 3 (1993): 225-262.

[3] Xu, Qiwen, Willem-Paul de Roever, and Jifeng He. *The rely-guarantee method for verifying shared variable concurrent programs.* Formal Aspects of Computing 9, no. 2 (1997): 149-174.

[4] Jones, Cliff B. *Wanted: a compositional approach to concurrency.* In Programming methodology, pp. 5-15. Springer, New York, NY, 2003.

[5] Vafeiadis, Viktor, and Matthew Parkinson. *A marriage of rely/guarantee and separation logic.* In CONCUR, vol. 4703, pp. 256-271. 2007.

[6] Gavran, Ivan, Filip Niksic, Aditya Kanade, Rupak Majumdar, and Viktor Vafeiadis. *Rely/guarantee reasoning for asynchronous programs.* In LIPIcs-Leibniz International Proceedings in Informatics, vol. 42. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[7] Reynolds, John C. *Separation logic: A logic for shared mutable data structures.* In Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on, pp. 55-74. IEEE, 2002.

[8] Parkinson, Matthew, Richard Bornat, and Peter O'Hearn. *Modular verification of a non-blocking stack.* In ACM SIGPLAN Notices, vol. 42, no. 1, pp. 297-302. ACM, 2007.

[9] Jacobs, Bart, and Frank Piessens. *Expressive modular fine-grained concurrency specification.* ACM SIGPLAN Notices 46, no. 1 (2011): 271-282.

[10] Sergey, Ilya, Aleksandar Nanevski, and Anindya Banerjee. *Mechanized verification of fine-grained concurrent programs.* In ACM SIGPLAN Notices, vol. 50, no. 6, pp. 77-87. ACM, 2015.

[11] Ohearn, Peter W. *Resources, concurrency, and local reasoning.* Theoretical computer science 375, no. 1-3 (2007): 271-307.

[12] Feng, Xinyu, Rodrigo Ferreira, and Zhong Shao. *On the relationship between concurrent separation logic and assume-guarantee reasoning.* In ESOP, vol. 7, pp. 173-188. 2007.

[13] Leino, K. Rustan M., and Peter Mller. *A basis for verifying multi-threaded programs.* In ESOP, vol. 9, pp. 378-393. 2009.

[14] Dinsdale-Young, Thomas, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. *Concurrent abstract predicates.* In ECOOP, vol. 6183, pp. 504-528. 2010.

[15] Jung, Ralf, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. *Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning.* ACM SIGPLAN Notices 50, no. 1 (2015): 637-650.

[16] Jung, Ralf, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. *Higher-order ghost state.* In ICFP, pp. 256-269. 2016.

[17] Dinsdale-Young, Thomas, Pedro da Rocha Pinto, Kristoffer Just Andersen, and Lars Birkedal. *Caper.* In European Symposium on Programming, pp. 420-447. Springer, Berlin, Heidelberg, 2017.

[18] Sergey, Ilya, JAMES R. WILCOX, and ZACHARY TATLOCK. *Programming and proving with distributed protocols.* Personal communication (2018).

[19] Hoare, Charles Antony Richard. *Communicating sequential processes.* In The origin of concurrent programming, pp. 413-443. Springer New York, 1978.

[20] Bergstra, Johannes Aldert, and Jan Willem Klop. *Fixed point semantics in process algebras.* (1982).

[21] Baeten, Jos CM. *A brief history of process algebra.* Theoretical Computer Science 335, no. 2-3 (2005): 131-146.

[22] Bolognesi, Tommaso, and Ed Brinksma. *Introduction to the ISO specification language LOTOS.* Computer Networks and ISDN systems 14, no. 1 (1987): 25-59.

[23] Milner, Robin. *A calculus of communicating systems.* (1980).

[24] Fidge, Colin. *A comparative introduction to CSP, CCS and LOTOS.* Software Verification Research Centre, University of Queensland, Tech. Rep (1994): 93-24.

[25] He, Jifeng, and Tony Hoare. *CSP is a retract of CCS.* In International Symposium on Unifying Theories of Programming, pp. 38-62. Springer, Berlin, Heidelberg, 2006.

[26] Abadi, Martin, Michael Burrows, Butler Lampson, and Gordon Plotkin. *A calculus for access control in distributed systems.* ACM Transactions on Programming Languages and Systems (TOPLAS) 15, no. 4 (1993): 706-734.

[27] Milner, Robin, Joachim Parrow, and David Walker. *A calculus of mobile processes, i.* Information and computation 100, no. 1 (1992): 1-40.

[28] Magee, Jeff, Naranker Dulay, Susan Eisenbach, and Jeff Kramer. *Specifying distributed software architectures.* Software EngineeringESEC'95 (1995): 137-153.

[29] Pierce, Benjamin C., and David N. Turner. *Pict: a programming language based on the Pi-Calculus.* In Proof, Language, and Interaction, pp. 455-494. 2000.

[30] Fournet, Cdric, and Georges Gonthier. *The reflexive CHAM and the join-calculus.* In Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 372-385. ACM, 1996.

[31] Honda, Kohei, Nobuko Yoshida, and Marco Carbone. *Multiparty asynchronous session types.* ACM SIGPLAN Notices 43, no. 1 (2008): 273-284.

[32] Castagna, Giuseppe, Mariangiola Dezani-Ciancaglini, Elena Giachino, and Luca Padovani. *Foundations of session types.* In Proceedings of the 11th ACM SIGPLAN conference on Principles and practice of declarative programming, pp. 219-230. ACM, 2009.

[33] Dezani-Ciancaglini, Mariangiola, and Ugo De'Liguoro. *Sessions and Session Types: An Overview.* In WS-FM, vol. 9, pp. 1-28. 2009.

[34] Gay, Simon J., and Vasco T. Vasconcelos. *Linear type theory for asynchronous session types.* Journal of Functional Programming 20, no. 1 (2010): 19-50.

[35] Caires, Lus, and Frank Pfenning. *Session Types as Intuitionistic Linear Propositions.* In CONCUR, vol. 10, pp. 222-236. 2010.

[36] Wadler, Philip. *Propositions as sessions.* ACM SIGPLAN Notices 47, no. 9 (2012): 273-286.

[37] Giunti, Marco, and Vasco Thudichum Vasconcelos. *Linearity, session types and the pi calculus.* In UNDER CONSIDERATION FOR PUBLICATION IN MATH. STRUCT. IN COMP. SCIENCE. 2013.

[38] Summers, Alexander J., and Peter Mller. *Actor Services.* In European Symposium on Programming Languages and Systems, pp. 699-726. Springer, Berlin, Heidelberg, 2016.

[39] Caires, Luis, Frank Pfenning, and Bernardo Toninho. *Linear logic propositions as session types.* Mathematical Structures in Computer Science 26, no. 3 (2016): 367-423.

[40] Desai, Ankush, Amar Phanishayee, Shaz Qadeer, and Sanjit A. Seshia. *Compositional Reasoning for Dynamic Distributed Systems.* (2017).

[41] Ancona, Davide, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Denilou, Simon J. Gay et al. *Behavioral types in programming languages.* Foundations and Trends in Programming Languages 3, no. 2-3 (2016): 95-230.

[42] Cardelli, Luca, and Andrew D. Gordon. *Mobile ambients.* In International Conference on Foundations of Software Science and Computation Structure, pp. 140-155. Springer, Berlin, Heidelberg, 1998.

[43] Abadi, Martn, and Andrew D. Gordon. *A calculus for cryptographic protocols: The spi calculus.* Proceedings of the 4th ACM conference on Computer and communications security. ACM, 1997.

[44] Smith, Howard. *Business process managementthe third wave: business process modelling language (bpml) and its pi-calculus foundations.* Information and Software Technology 45.15 (2003): 1065-1069.

[45] Svendsen, Kasper, and Lars Birkedal. *Impredicative Concurrent Abstract Predicates.* In ESOP, vol. 8410, pp. 149-168. 2014.

[46] da Rocha Pinto, Pedro, Thomas Dinsdale-Young, and Philippa Gardner. *TaDA: A logic for time and data abstraction.* In European Conference on Object-Oriented Programming, pp. 207-231. Springer, Berlin, Heidelberg, 2014.

[47] Luttik, Bas. *What is algebraic in process theory?.* Electronic Notes in Theoretical Computer Science 162 (2006): 227-231.

[48] Hatzel, Meike, Christoph Wagner, Kirstin Peters, and Uwe Nestmann. *Encoding CSP into CCS.* arXiv preprint arXiv:1508.06712 (2015).

[49] Bhargavan, Karthikeyan, Ricardo Corin, Pierre-Malo Denilou, Cdric Fournet, and James J. Leifer. *Cryptographic protocol synthesis and*

verification for multiparty sessions. In Computer Security Foundations Symposium, 2009. CSF'09. 22nd IEEE, pp. 124-140. IEEE, 2009.

[50] Caires, Lus, Jorge A. Prez, Frank Pfenning, and Bernardo Toninho. *Behavioral Polymorphism and Parametricity in Session-Based Communication.* In ESOP, vol. 7792, pp. 330-349. 2013.

[51] Httel, Hans, Ivan Lanese, Vasco T. Vasconcelos, Lus Caires, Marco Carbone, Pierre-Malo Denilou, Dimitris Mostrous et al. *Foundations of session types and behavioural contracts.* ACM Computing Surveys (CSUR) 49, no. 1 (2016): 3.