Bridging the Cyber and Physical with a Verifiable, Executable Language for Robotics

Jiawei Chen[1] , José Luiz Vargas de Mendonça[1] , and Jean-Baptiste Jeannin[1]

***Abstract*— Building reliable Cyber-Physical Systems (CPS) often requires formal verification, which is capable of producing the rigorous safety guarantees necessary to provide confidence in its design. However, verification may be challenging to perform on real-world systems, and as a result, a simplified model may be verified instead. This results in a disconnect between verification and reality, leading to diminished confidence in the verification result. We have developed a method that enables CPS designers to both formally verify and implement CPS software in a single language. By combining the industry-tested modeling and compilation of synchronous programming with the verification rigor of refinement types, we strive to give CPS designers additional confidence in designing reliable, real-world CPS.**

I. Introduction

Designing high-reliability CPS often entails formal verification, but this method requires that the system being verified is an accurate representation of the real system. Unfortunately, this is not always straightforward. Programming languages for CPS design require well-understood formal semantics for verification and expressiveness for producing implementation-level software, but there are challenges in reconciling the two. Real implementations of CPS may be complex and difficult to characterize, making verification difficult unless performed on an abstract model. This results in uncertainty in the validity of the verification. We present a method for verification and implementation in a single language, by combining the industry-tested modeling and compilation capabilities of synchronous programming with formal verification through an extended type system. With this method, we have formally verified and implemented an autonomous vehicle software component on a real robot [1].

A. Synchronous Modeling and Execution

Synchronous programming languages treat CPS as collections of functions that operate on streams of incoming data, such as sensor values, to produce output streams, such as actuator commands [2], [3]. By design, synchronous programs have limited memory and must react on the same time base as their inputs, which makes them useful for modeling embedded systems. Implementations of synchronous languages, such as Scade [4] and Esterel [5], have already been employed in industry, modeling critical systems such as avionics and integrated circuits [6]. We extend Zélus, an existing Lustre-based synchronous programming language already capable of producing executable simulations [7], [8], with verification and the ability to operate real robots.

[1]University of Michigan, Ann Arbor, MI, USA
{chenjw, joselvdm, jeannin}@umich.edu

B. Verification through Refinement Types

Formal verification is introduced via refinement types, in the style of Liquid Haskell [9], [10]. Refinement types extend a language's type system by allowing type specifications to refer to program values. For instance, a value that must only be a positive integer may be specified as $\{v : \mathsf{int} \mid v > 0\}$, where v is a variable that refers to terms of this type. Values of the type can be any integer v where $v > 0$. The incorporation of logical predicates into type specifications and the ability to refer to program terms make refinement types a powerful specification tool. Refinement types also inherit some desirable properties of ordinary types, such as compile-time verification and modularity [9]. Although type-checking has already been applied to safety-critical verification tasks, such as those in robotics [11] and networking [12], extending refinement type checking to streams presents some unique challenges, which we address in our project. Our contribution includes the translation of refinement types in synchronous programs into constraints that can be checked automatically, such as with the Z3 SMT solver [13].

C. Related Work

Hybrid automata [14] and differential dynamic logic [15] are commonly used for modeling and verifying CPS. Although they can model real-world systems [16], their non-deterministic nature precludes directly generating executable code, though translation may be possible [17]. Nevertheless, there exists a gap between the CPS that is implemented on a real system and the model CPS that is verified.

Synchronous program verification has also been addressed in existing projects, such as Kind [18] and Kind 2 [19], which use an approach similar to bounded model checking to synthesize invariants for Lustre programs. Similarly, CoCoSim [20] uses Simulink as an interface to build synchronous programs with specifications that are translated to Lustre to be verified. Our approach differs in that the constraints generated for type checking allows for a more thorough search of the state space, but may require invariants to avoid spurious counterexamples.

Verifiable and executable CPS have been explored in other works, such as VeriPhy [17] and Koord [21], [22]. VeriPhy proposes a method for compiling differential dynamic logic specifications into runtime monitors that allow a system to recover from anomalies [17]. Koord allows for verification of coordination-level multi-agent robotics systems and the implementation of programs on real robots. Though similar in use case, our project fulfills a distinct role relative to these existing works. For instance, one may use our work to design

and verify the fallback controller used in VeriPhy, or the platform-specific controllers used in Koord.

II. Methods

We chose to extend the Zélus language [7], due to its extensive existing work in the implementation and compilation of synchronous programs [8]. Additionally, its support for hybrid systems allows one to model systems with continuous-time dynamics, though we note that our current verification work extends only to the discrete-time domain. We base the extended type system on that of Liquid Haskell [10], whose translation of type specifications into automatically solvable constraints is straightforward and can be readily adapted to synchronous programs. We then bridge the gap between the synchronous program and the real world using a protocol reminiscent of CyPhyHouse [22]. Combining these components, we ensure a seamless workflow from verification to execution in a single language. Technical details regarding the language and implementation can be found in our workshop paper [1].

A. Type-Checking Synchronous Programs

We developed a set of typing rules modeled after existing refinement type systems [9], [10], [23]. A unique challenge that arises from combining synchronous programs with refinement types is ensuring that types on streams indeed describe all possible values of the stream. Unlike ordinary refinement types, our type system must also account for temporal properties. For example, a branching instruction such as an if-statement may now be conditioned on a stream of boolean values. As a result, the program repeatedly switches between the two branches, which themselves exhibit temporal behavior. We note that, in the discrete subset of Zélus, streams can be modeled inductively, with one or more initial conditions and an evolution function that determines the stream's subsequent value using a finite number of previously-computed values. Using this property, universal temporal properties, specifying that something always or never happens, can be conveniently encoded as SMT constraints, though inductive invariants may be necessary to rule out false counterexamples. Although only a subset of temporal logic, many safety properties of note can be encoded as a universal temporal property in practice. The typing rules we developed form the foundation of our type checker, which serves as an interface between the Zélus program augmented with refinement types and the Z3 SMT solver used to verify that all refinements are properly satisfied.

B. Verifying a Real-World Example

We demonstrated the design process of a simple robot program modeled after a vehicle's adaptive cruise control system [16]. In the various scenarios we sought to verify, the robot with the formally-verified controller (the "ego" vehicle) is to follow another vehicle along a single lane, which may choose to accelerate or decelerate without the knowledge of the ego vehicle. The ego vehicle is specified to avoid a collision at all times. For simplicity, the ego vehicle's controller is a bang-bang controller which can only either accelerate or brake at its maximum rate. However, we note that more complex controllers can also be verified using this method, given the appropriate invariants are chosen. The key insight behind choosing invariants often entails explicitly defining an inductive property of the system. For instance, the inductive invariant we employed to verify adaptive cruise control involved assuming that the controller in the last time step did not put the vehicle into an unsafe configuration in the current time step. The verification task then becomes proving that the controller in the current time step continues to make a safe decision. Combined with verifying that the system began in a safe configuration, the safety property can be proven inductively.

C. Executing Synchronous Programs on Real Robots

Although the Zélus compiler in its unmodified state is able to generate executable simulations, these simulations have no interactions with the physical world, save for synchronizing simulations to real time. We extend Zélus with an inter-process communications scheme based on Lightweight Communications and Marshalling (LCM) [24] to share variable values between the synchronous program and low-level drivers for various robot peripherals. The low-level drivers implement simple motor speed controllers and sensor sampling functionality for convenience, though it is possible to implement parts of these in the verified synchronous code to shrink the trusted base. Our current experiments employ a small differential-drive wheeled robot equipped with LiDAR and wheel speed sensors, making it well-suited for simulating autonomous vehicles. We have published experimental results from a simplified version of the aforementioned adaptive cruise control scenario [1], and we have results from the full system in submission.

III. Discussion and Future Work

The project's primary goal is to develop a trusted chain from system specification and modeling to implementation on real-world CPS. Our methods extend the modeling capabilities of synchronous languages with specifications through type annotations, allowing for compile-time verification of safety properties. The present work trusts that the compiler correctly translates the verified synchronous program into safe executables. There have been developments in formally-verified compilation of synchronous languages [25], which if implemented would further enhance trust in our methods. We also assume that the user has correctly specified the environment in which the CPS is to operate. Since we employ compile-time verification, we carry assumptions made about the environment's behavior at runtime. A possible solution could be to incorporate run-time monitoring, similarly to VeriPhy [17] or RTLola [26], which will be able to use our generated verification constraints to check that these assumptions are met. Finally, we would like to extend verification to hybrid components of the Zélus language, giving the user even more expressiveness in verifying and implementing safe CPS.

REFERENCES

[1] J. Chen, J. L. Vargas de Mendonça, S. Jalili, B. Ayele, B. N. Bekele, Z. Qu, P. Sharma, T. Shiferaw, Y. Zhang, and J.-B. Jeannin, "Synchronous Programming and Refinement Types in Robotics: From Verification to Implementation," in *Proceedings of the 8th ACM SIGPLAN International Workshop on Formal Techniques for Safety-Critical Systems*, FTSCS 2022, (New York, NY, USA), pp. 68–79, Association for Computing Machinery, Dec. 2022.

[2] N. Halbwachs, *Synchronous Programming of Reactive Systems*. Boston, MA: Springer US, 1993.

[3] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice, "LUSTRE: A declarative language for programming synchronous systems," in *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages - POPL '87*, (Munich, West Germany), pp. 178–188, ACM Press, 1987.

[4] J.-L. Colaço, B. Pagano, and M. Pouzet, "SCADE 6: A formal language for embedded critical software development (invited paper)," in *2017 International Symposium on Theoretical Aspects of Software Engineering (TASE)*, pp. 1–11, Sept. 2017.

[5] F. Boussinot and R. de Simone, "The ESTEREL language," *Proceedings of the IEEE*, vol. 79, pp. 1293–1304, Sept. 1991.

[6] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone, "The synchronous languages 12 years later," *Proceedings of the IEEE*, vol. 91, pp. 64–83, Jan. 2003.

[7] A. Benveniste, T. Bourke, B. Caillaud, and M. Pouzet, "A hybrid synchronous language with hierarchical automata: static typing and translation to synchronous code," in *Proceedings of the ninth ACM international conference on Embedded software - EMSOFT '11*, (Taipei, Taiwan), p. 137, ACM Press, 2011.

[8] T. Bourke, J.-L. Colaço, B. Pagano, C. Pasteur, and M. Pouzet, "A Synchronous-Based Code Generator for Explicit Hybrid Systems Languages," in *Compiler Construction* (B. Franke, ed.), vol. 9031, pp. 69–88, Berlin, Heidelberg: Springer Berlin Heidelberg, 2015. Series Title: Lecture Notes in Computer Science.

[9] P. M. Rondon, M. Kawaguci, and R. Jhala, "Liquid types," in *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation - PLDI '08*, (Tucson, AZ, USA), p. 159, ACM Press, 2008.

[10] N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton-Jones, "Refinement types for Haskell," in *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*, ICFP '14, (New York, NY, USA), pp. 269–282, Association for Computing Machinery, Aug. 2014.

[11] A. Anand and R. Knepper, "ROSCoq: Robots Powered by Constructive Reals," in *Interactive Theorem Proving* (C. Urban and X. Zhang, eds.), vol. 9236, pp. 34–50, Cham: Springer International Publishing, 2015. Series Title: Lecture Notes in Computer Science.

[12] M. Eichholz, E. H. Campbell, M. Krebs, N. Foster, and M. Mezini, "Dependently-typed data plane programming," *Proceedings of the ACM on Programming Languages*, vol. 6, pp. 1–28, Jan. 2022.

[13] L. de Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *Tools and Algorithms for the Construction and Analysis of Systems* (D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, C. R. Ramakrishnan, and J. Rehof, eds.), vol. 4963, pp. 337–340, Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. Series Title: Lecture Notes in Computer Science.

[14] R. Alur, C. Courcoubetis, T. A. Henzinger, and P. H. Ho, "Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems," in *Hybrid Systems* (G. Goos, J. Hartmanis, R. L. Grossman, A. Nerode, A. P. Ravn, and H. Rischel, eds.), vol. 736, pp. 209–229, Berlin, Heidelberg: Springer Berlin Heidelberg, 1993. Series Title: Lecture Notes in Computer Science.

[15] A. Platzer, "Differential Dynamic Logic for Hybrid Systems," *Journal of Automated Reasoning*, vol. 41, pp. 143–189, Aug. 2008.

[16] S. M. Loos, A. Platzer, and L. Nistor, "Adaptive Cruise Control: Hybrid, Distributed, and Now Formally Verified," in *FM 2011: Formal Methods* (M. Butler and W. Schulte, eds.), vol. 6664, pp. 42–56, Berlin, Heidelberg: Springer Berlin Heidelberg, 2011. Series Title: Lecture Notes in Computer Science.

[17] R. Bohrer, Y. K. Tan, S. Mitsch, M. O. Myreen, and A. Platzer, "VeriPhy: verified controller executables from verified cyber-physical system models," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, (Philadelphia PA USA), pp. 617–630, ACM, June 2018.

[18] G. Hagen and C. Tinelli, "Scaling Up the Formal Verification of Lustre Programs with SMT-Based Techniques," in *2008 Formal Methods in Computer-Aided Design*, pp. 1–9, Nov. 2008.

[19] A. Champion, A. Mebsout, C. Sticksel, and C. Tinelli, "The Kind 2 Model Checker," in *Computer Aided Verification* (S. Chaudhuri and A. Farzan, eds.), vol. 9780, pp. 510–517, Cham: Springer International Publishing, 2016. Series Title: Lecture Notes in Computer Science.

[20] H. Bourbouh, P.-L. Garoche, T. Loquen, Noulard, and C. Pagetti, "CoCoSim, a code generation framework for control/command applications An overview of CoCoSim for multi-periodic discrete Simulink models," in *10th European Congress on Embedded Real Time Software and Systems (ERTS 2020)*, (Toulouse, France), Jan. 2020.

[21] R. Ghosh, C. Hsieh, S. Misailovic, and S. Mitra, "Koord: a language for programming and verifying distributed robotics application," *Proceedings of the ACM on Programming Languages*, vol. 4, pp. 1–30, Nov. 2020.

[22] R. Ghosh, J. P. Jansch-Porto, C. Hsieh, A. Gosse, M. Jiang, H. Taylor, P. Du, S. Mitra, and G. Dullerud, "CyPhyHouse: A programming, simulation, and deployment toolchain for heterogeneous distributed coordination," in *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 6654–6660, May 2020. ISSN: 2577-087X.

[23] R. Jhala and N. Vazou, "Refinement Types: A Tutorial," *arXiv:2010.07763 [cs]*, Oct. 2020. arXiv: 2010.07763.

[24] LCM Project, "Lightweight Communications and Marshalling."

[25] T. Bourke, L. Brun, P.-E. Dagand, X. Leroy, M. Pouzet, and L. Rieg, "A Formally Verified Compiler for Lustre," *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 586–601, 2017.

[26] J. Baumeister, B. Finkbeiner, S. Schirmer, M. Schwenger, and C. Torens, "RTLola Cleared for Take-Off: Monitoring Autonomous Aircraft," in *Computer Aided Verification* (S. K. Lahiri and C. Wang, eds.), vol. 12225, pp. 28–39, Cham: Springer International Publishing, 2020. Series Title: Lecture Notes in Computer Science.