Cypress

A Testbed for Research in Networked Cyber-Physical Systems

Ryan Goodfellow
USC Information Sciences Institute
Marina del Rey, CA
rgoodfel@isi.edu

Erik Kline
USC Information Sciences Institute
Marina del Rey, CA
kline@isi.edu

ABSTRACT

In this paper we present the progress of our current work in the design and implementation of a cyber-physical system (CPS) testbed called Cypress. The purpose of Cypress is to provide a high-fidelity testbed environment that cohesively integrates the cyber and physical domains across all phases of the scientific experimentation process. Cypress is composed of a language, compiler, simulation engine and distributed control runtime system facilitating the development, execution and analysis of CPS experiments. Cypress is primarily focused on CPSs that may be classified as distributed or networked control systems.

Through its declarative experiment description language, Cypress presents a gradient of usability accessible to both engineers from physical domains as well as those from computer science. The language captures physical, control, computing and communication models. The compiler translates an experiment codebase into an experiment package comprised of a set of simulation executables, a network environment description and distributed control software. The runtime system instantiates and orchestrates the execution of experiment packages by creating the network testbed environment, deploying control system software within that environment and managing the execution of the simulation. Cypress is an extension to the DeterLab [1, 12] network testbed environment and leverages many of the facilities already provided within that ecosystem.

General Terms

Experimentation, Languages

Categories and Subject Descriptors

I.6.0 [**Simulation and Modeling**]: General; J.6 [**Computer Aided Engineering**]

1. INTRODUCTION

Cyber-physical system development is by definition an interdisciplinary endeavor. In this work we focus specifically on the development of networked distributed control systems. The basic motivation for Cypress is to create a testbed environment where the cyber-physical system under experimentation is treated holistically from experiment expression to deployment and execution in a combined simulated-emulated environment. Currently, the typical workflow in designing such systems is to model the physical system under control, formulate a set of controllers capable of achieving the desired objectives, implement the distributed control in an environment where it can be tested and then translate that implementation into something that can be deployed into the real world. In each phase of this process there is an increasing breadth of complexity that must be dealt with by systems designers. Theoretical models are fantastic at isolating and analyzing the core capabilities that a system must provide, but isolation fades as the model is transitioned closer to the real world and fidelity increases. This is particularly true for developing *distributed* control applications where software must be built which has significant complexity at both the distributed computation level and the networked system level. By treating the experimentation process holistically, the testbed environment can provide tools that help to both manage and understand interdomain cyber-physical complexities.

There is currently no holistic approach to conducting CPS experimentation at a level of fidelity which is a reasonable proxy to a real-world networked control environment. Theoretical frameworks exist in which distributed control systems may be formulated as distributed computations [11], but how properties in theory translate to practice still remains an open question. Cypress provides a means by which to rigorously experiment with distributed control systems at the implementation level across cyber-physical boundaries through the combination of its expression language, compiler, simulator and runtime system.

This paper is organized as follows. We begin in §2 by presenting the workflow for CPS experiment design, execution and analysis that Cypress enables. The high-level architecture of the testbed facilities is overviewed in §3. The current capabilities are then discussed in §4 followed by future work in §5.

2. WORKFLOW

The Cypress workflow is designed for experiment driven development of CPSs. The workflow is illustrated in Figure 1. The dotted lines indicate the design phase of an experiment, the dashed lines indicate the execution phase, and the intermixed dotted and dashed lines represents a combination of the two.

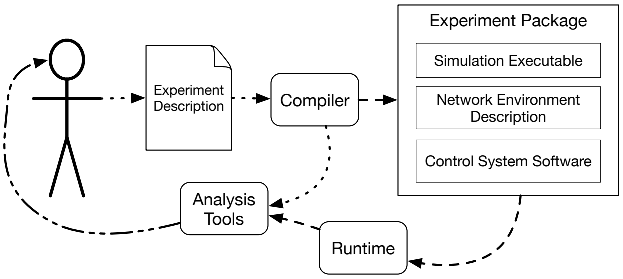

Figure 1: The Cypress Workflow

2.1 Design

The development lifecycle begins with the experiment expression language and the compiler. Like development in most programming languages, this is an iterative process between the designer and the compiler where the compiler is performing static analysis on the experiment code and giving feedback to the user. Cypress provides analysis tools which can be used in combination with the compiler. The compiler covers basic syntax and semantics of an experiment codebase while the analysis tools have both broader and deeper system-level CPS static analysis capabilities. The code in Figure 2 is a very simple Cypress experiment. There are three principle kinds in Cypress; Objects (line 2), Controllers (line 7) and Experiments (line 11). We are using the word *kind* in the formal language sense here e.g., a higher order type that is by its very nature extensible through a composition of lower level language elements. An instance of a kind is a type of which we see 3 examples in this code; a Rotor is a type that is of kind Object, the RotorSpeedController is (redundantly) a Controller and RotorControl is an experiment. It may seem that we are overloading the word experiment to refer to both this code file in its entirety as well as the actual Experiment element. In actuality, a compilation unit in Cypress is defined to have one and only one experiment, somewhat like the main method in C, so we just refer to the whole thing as an experiment.

```
1   //Cypress Rotor Controller Experiment
2   Object Rotor(H)
3     ω = θ'
4     a = τ - H*ω^2
5     a = ω'
6
7   Controller RotorSpeedController(ωt)
8     a' = ωt - ω
9     τ = a
10
11  Experiment RotorControl
12    Rotor rotor(H:2.5, ω|0, θ|0)
13    RotorSpeedController ctrl(ωt:100, τ|2)
14    Link lnk0(Latency:5, Bandwidth:100)
15    Link lnk1(Latency:10, Bandwidth:100)
16
17    rotor.ω > |0.01| > lnk0 > ctrl.ω
18    ctrl.τ > lnk1 > rotor.τ
```

Figure 2: The Cypress Language

Cypress is an indent delimited language much like Python, so anything below an instance of a kind that is indented further than a type declaration is code that belongs to that declaration. Cypress draws heavily from Modelica [4]. It is modular language where the Objects and Controllers are defined as compositions of differential-algebraic equations and Experiments are compositions of Objects, Controllers and the connections between them. In this particular example, the Rotor object is a simple abstract model with an angular position θ, angular velocity ω, applied torque τ, inertial constant H and angular acceleration a. The RotorSpeedController attempts to control these properties by modulating acceleration to keep the actual angular velocity in line with the target angular velocity ωt.

The experiment definition begins with component instances. In the first instance called rotor, there are both parameters and initial values applied to the definition of the instance. The colon operator which is infix between H and 2.5 indicates parameter assignment. This parameter assignment references back to the Rotor type declaration at line 2. All parameters that are listed in a type declaration must be passed as parameters at instantiation. The pipe operator which is infix between ω and 0 indicates initial value initialization and is optional. If type values are not given, Cypress will attempt to create a complete set of consistent initial conditions from the partial (or null) set that is given.

The second part of the experiment definition is the interconnection of components. This is shown in lines 17-18 of Figure 2. From left to right on line 17: $rotor.\omega$ is a reference to the angular velocity component of the rotor instance at line 12. This is then supplied to the infix $>$ operator, the second argument of which is an analog to digital (AtoD) literal expression. The number inside the AtoD is the rate at which the physical quantity $rotor.\omega$ is sampled into a digital stream of values. In this case 100 times per second. That digital stream is then fed to $lnk0$ which refers to a data link with a latency of 5 milliseconds and bandwidth of 100 mbps. The stream of sampled values then passes through the link to its final destination $ctrl.\omega$, the angular velocity parameter of the RotorSpeedController. The following line takes the torque value computed by the controller and sends its sampled value representation through link1 to control the torque value on the rotor.

When the Cypress compiler compiles this file, the result is 1) a physical simulation executable that simulates the rotor object, 2) C++ source and compiled executable for the RotorSpeedController, 3) a network topology description file which specifies the network testbed environment in which the simulator and control software execute and, 4) some scripting interface files to launch the experiment package in the Cypress testbed environment.

2.2 Execution and Analysis

When an experiment package is submitted to the runtime system, the following takes place. Based on the network description file, a DeterLab network environment is created. It is within the DeterLab runtime environment that the entire Cypress runtime system exists. A testbed node is created for each controller in the experiment and links between controllers and other controllers or the simulator are created per the description file. Additionally, there are special nodes with particular compute capabilities that are reserved as simulation nodes. There will be more on the architecture in §3. Once the network environment portion of the experiment has been established, the control node software and simulation software is deployed to the corresponding nodes. When deployment is complete the experiment enters the ready state. The scripting interface files that were generated as a part of the experiment package may then be used to start, monitor, and collect the results of the experiment.

The scripting interface files are built on top of the Magi experiment orchestration framework [7] that already exists as a part of the DeterLab ecosystem. Magi is an experiment lifecycle management tool base. Using Magi, the Cypress interface scripts can coordinate the startup of controllers with the physical simulator as well as automate the retrieval of results. From a workflow perspective, this has the advantage of automating tasks which need not burden experimenters. Moreover, distributed coordination can be fairly difficult to get right. From a scientific experimentation perspective, employing a uniform mechanism for these tasks promotes reproducibility.

Experiment results are stored in an experiment database. For the class of experiments Cypress supports, we are primarily concerned with the time series evolution of the physical variables under control. This time series data is automatically stored for every experiment. If more information is required by the experimenter there is a query and set interface within the runtime system to enable additional outputs. Such outputs can range from physical variables that are not under control to the dynamic properties of the network such as link or control node processor loading. Magi is also instrumental in supporting these capabilities. The runtime environment employs Magi agents to coordinate the movement of result data into the experiment database and ultimately back to the user as it becomes available.

3. ARCHITECTURE

The architecture of Cypress may be considered as a layer on top of DeterLab. DeterLab provides the foundation and Cypress provides extensions for enabling cyber-physical experimentation. This is depicted in the top portion of Figure 3, the bottom half of the figure shows the hardware resources that are utilized though this architecture. The execution control, package deployment, i/o control and data storage models were introduced in §2.2. Execution control and data storage depend on the DeterLab orchestration facilities. Package deployment depends on DeterLab resource allocation and isolation as well as virtualization. In many cases the individual pieces of control software are quite simple and demand very little in terms of resources. Thus, virtualization of control nodes helps achieve scale and isolation at the same time. The DeterLab container [3] system provides a range of virtualization options which present tradeoffs between scalability and fidelity. This affords flexibility to the package deployment module in mapping the combination of the abstract topology and distributed control software generated by the compiler to an experiment instance running on the testbed. The distributed control software carries metadata which is generated by the compiler to help the deployment module in synthesizing an appropriate embedding.

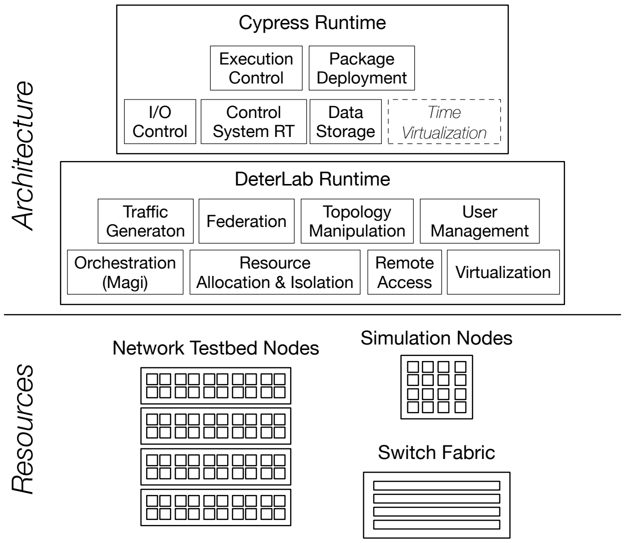

Figure 3: The Cypress-DeterLab Architecture

The control system module is a runtime system component library. The distributed control software generated by the experiment compiler and the physical simulator use the library to communicate at experiment execution time. The module is a data-decoupled publish/subscribe system which allows controllers to propagate control signals to the simulator and for the simulator to propagate sampled physical values back to controllers. In publish/subscribe parlance, there are two types of topics — control topics and measurement topics. Control topics are published by controllers and subscribed to by the simulator. Measurement topics are published by the simulator and subscribed to by controllers. The use of publish/subscribe-style communication through middleware is well established in many control systems communities [13, 10]. As future work, we plan to adopt an accepted controls protocol such as OMG-DDS to facilitate an easy transition path from generated testbed software to real world software.

4. CURRENT STATUS

This section describes the current capabilities and limitations of the Cypress testbed. This is best described by the types of CPSs the language can express and, the scale and fidelity at which the runtime system is able to support their execution. Development tasks to remove all of the limitations discussed are under consideration for future releases. We begin with expression.

The Cypress language provides the following expressive capabilities.

1. DAE based physical and controller modeling.
2. Experiment modeling
   (a) Physical, controller and network link model instantiation.
   (b) Sampling rate specification.
   (c) Component interconnection.

A substantial limitation on network expression is the restriction to point-to-point link models. DeterLab is capable of emulating almost arbitrarily complex IP networks. However, the Cypress language is only able to express network topologies composed of point-to-point inter-controller and controller-simulator links. DeterLab also provides mechanisms to modify link parameters at runtime, however there is no mechanism at the current time to create such dynamics within a Cypress experiment. Experiments that require varying network conditions, must be recompiled and rerun or have their network parameters manually altered.

Control system models, much like the physical system models, are defined as systems of interdependent DAEs. Each controller has a set of local and remote variables it uses to compute the solution to a time parametric problem whose timescale is based on the input/output rate of the controller. At present time, this input/output rate is dictated by the AtoD elements within the experiment. This means that the input/output rate across controllers is uniform and cannot support for example, sliding or hopping window based controls. This also means that algorithmically defined controls are not possible at this time.

Controllers are implemented as distributed software that is generated by the compiler. This software uses Sundials [6] for computing solution values of output variables. The controllers do this in much the same way as the distributed MPI based solvers in the simulator. Except in this case data distribution is carried out by the publish subscribe framework and communication takes place over the experiment network as opposed to a cluster computing network. This allows for communications phenomena to affect the performance of the control system. Each control node sees its own local version of the state space based on what information it has received from other controllers and from the simulator. If this information is either not current or erroneous, it will be reflected in the output of the controller. This opens up the door to experiments that asses traffic generation types attacks as well as data injection/falsification attacks.

For physical system simulation, the testbed supports nonlinear differential-algebraic equation (DAE) based models. The simulation engine is based on Sundials [6] and was built specifically for Cypress. The compiler translates physical object descriptions into a distributed multiple-program-multiple-data (MPMD) message passing interface (MPI) based simulation. This provides a scalable means by which to compute the residual of the system DAE. The residual is then passed back to the Sundials IDA solver through the MPI based vector data interface provided by the library.

Currently the simulator operates in real-time mode. This simplifies the implementation of the runtime system for the controllers because there is no artificial synchronization that needs to take place. However, this severely limits scalability as we are stuck with physical systems that we can simulate in real time. Also there is currently no a priori way to tell whether or not a simulation can be run in real time. If the real-time constraint is indeed broken, this is detected by the simulator and reported to the experimenter.

The experiment compiler and analytical tools provide users with feedback on experiment code which catch modeling errors, violations in assumptions and assist in design time tasks such as establishing initial conditions for the physical system. Interactive design time analytics are essential for building large scale experiments. The analytic capabilities are built as a library of algorithms that run over the experiment abstract syntax tree generated by the compiler. These libraries are built for extensibility by users of the system. The current built-in capabilities for static experiment analysis are still quite rudimentary. The obvious semantics that a compiler must enforce such as making sure connections are well defined and instantiation parameters supplied are in place. For system level tools we provide a global balance checker e.g., ensuring the number of equations equal to the number of unknowns in the physical simulation. We also provide consistency and singular systems checks with respect to initial conditions.

We are actively developing a power systems library using the Cypress language. At the time of writing, this library is still in the very early stages. We are using the development of this library as a vehicle for understanding the requirements of the language as well as what analytical tools are helpful in developing large scale CPS experiments. Scalability of the simulation engine and control system runtime environment is also planned to be assessed using the family of experiments that grow out of this library.

5. FUTURE WORK

Development of Cypress is at an early stage, with significant ongoing work in progress. In addition to addressing expressive limitations within the current modeling language, as discussed in §4, we are exploring a number of entirely new capabilities for the system. We discuss two areas of active development below: extensible communications protocol support and virtualized time management.

The current controller communication mechanism is a thin pub-sub abstraction layer on top of MPI group-based communication. For real control systems there are many purpose built protocols and communications libraries. In some cases the designer of a control system may have little to no choice in protocol selection. Large systems may have many control subsystems running a heterogeneous spectrum of protocols. The interaction between subsystems can impact the overall performance and security of the system. Thus, it is important for a testbed that is designed for pragmatic CPS experimentation to support the use of protocols which are relevant to the systems under test. Ideally this should be accomplished through extensibility and not provision. Expertise in experiment specific protocols clearly belongs to the experimenters. The testbed will aim to provide a plugin type mechanism to support user supplied protocols and communication libraries.

A key problem to solve when linking a simulator to an emulation environment is that simulators will speed up or slow down time based on current resource utilization or to control accuracy. An emulator must be responsive to the adjustments the simulator makes in order for the entire experiment to maintain validity. Significant work has been conducted in the realm of static time virtualization. UCSD's Die Cast system [5] allows for multiple systems to share a constant time dilation factor while correctly handling the complex interactions between the dilated systems and the physical connections between them. They also solve many of the complex internal problems described such as scheduling and I/O. Die Cast and Zheng et al. [9] also allow for dynamic time adjustments but only in the case of all virtual systems existing on a single physical system. Simultaneously synchronizing time management dynamically across a distributed system is an open problem. Initially, we will begin with a statically synchronized simulation-emulation environment based on worst case run time analysis. We plan to build significantly upon this to develop a fully dynamic time virtualized distributed system. Simulators can also revert to previous time epochs when errors are detected. We will enhance our emulation environment to also allow time reversion based on Lamport's snapshot algorithm [2] and Time Warp [8].

6. CONCLUSION

This paper has introduced Cypress, a testbed for CPS distributed control experimentation. Cypress provides researchers from a diverse spectrum of the CPS community with a testbed resource that presents an approachable yet scalable gradient of usability thus facilitating rapid experimentation and development. Cypress is the first testbed that we know of that brings together modeling, analysis and experimentation of networked CPS control systems.

7. REFERENCES

[1] T. Benzel. The science of cyber security experimentation: the DETER project. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pages 137–148. ACM, 2011.

[2] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, Feb. 1985.

[3] DETER Team. Building apparatus for multi-resolution networking experiments using containers. Technical Report ISI-TR-683, Information Sciences Institute, 2011.

[4] P. Fritzson and V. Engelson. Modelica: A unified object-oriented language for system modeling and simulation. In *ECOOP'98 Object-Oriented Programming*, pages 67–90. Springer, 1998.

[5] D. Gupta, K. V. Vishwanath, M. McNett, A. Vahdat, K. Yocum, A. Snoeren, and G. M. Voelker. Diecast: Testing distributed systems with an accurate scale model. *ACM Trans. Comput. Syst.*, 29(2):4:1–4:48, May 2011.

[6] A. C. Hindmarsh, P. N. Brown, K. E. Grant, S. L. Lee, R. Serban, D. E. Shumaker, and C. S. Woodward. Sundials: Suite of nonlinear and differential/algebraic equation solvers. *ACM Transactions on Mathematical Software (TOMS)*, 31(3):363–396, 2005.

[7] A. Hussain and J. Chen. Montage topology manager: Tools for constructing and sharing representative internet topologies. Technical Report ISI-TR-684, Information Sciences Institute, 2012.

[8] D. R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7:404–425, 1985.

[9] D. Jin, Y. Zheng, and D. M. Nicol. A parallel network simulation-emulation testbed. *J Simulation*, 8(3):206–214, Aug 2014.

[10] W. Kang, K. Kapitanova, and S. H. Son. Rdds: A real-time data distribution service for cyber-physical systems. *Industrial Informatics, IEEE Transactions on*, 8(2):393–405, 2012.

[11] D. K. Kaynar, N. Lynch, R. Segala, and F. Vaandrager. The theory of timed i/o automata. *Synthesis Lectures on Distributed Computing Theory*, 1(1):1–137, 2010.

[12] J. Mirkovic, T. Benzel, T. Faber, R. Braden, J. Wroclawski, and S. Schwab. The DETER project: Advancing the science of cyber security experimentation and test. In *Technologies for Homeland Security (HST), 2010 IEEE International Conference on*, pages 1–7, Nov 2010.

[13] G. Pardo-Castellote. Omg data-distribution service: Architectural overview. In *Distributed Computing Systems Workshops, 2003. Proceedings. 23rd International Conference on*, pages 200–206. IEEE, 2003.