Cyber-Physical Verification of Intermittently Powered Embedded Systems

Rose Bohrer, *Member, IEEE*[*] and Bashima Islam, *Member, IEEE*[†]
Worcester Polytechnic University
Worcester, MA, USA
Email: [*]rbohrer@wpi.edu, [†]bislam@wpi.edu

***Abstract*—Intermittently powered embedded systems are a foundational and growing component of the Internet-of-Things. It is essential to rigorously prove these systems' correctness because they arise both in safety-critical applications and applications where quality-of-service is essential to social good. Such proofs are challenging because they are simultaneously cyber-physical and time-sensitive: correctness is affected by physical properties that change with time. This paper introduces a new general-purpose formal verification approach for cyber-physical properties of intermittent systems. We define a high-level modeling and specification language for intermittent systems, define its formal semantics, and prove that the language reduces to hybrid games, enabling application of existing theorem-proving software. Cold storage for COVID vaccines serves as a running example; we provide a machine-checked proof that safe temperatures are maintained under suitable assumptions. The crux of our proof approach is to identify power and timing assumptions under which sufficient power is available to complete time-sensitive tasks. Orthogonal to approaches that prove new guarantees on power or timing, our work rigorously shows which power and timing assumptions are needed for cyber-physical correctness.**

***Index Terms*—Intermittent computing, cyber-physical systems, real-time systems, programming language semantics.**

I. Introduction

INTERMITTENTLY powered embedded systems compute using ambient energy (e.g., solar power, thermal energy, wind energy, salinity gradients, and kinetic energy). These systems are called **Intermittent Computing Systems (ICSs)**. ICSs are the stepping stone for making everyday objects intelligent and unleashing the full potential of the Internet-of-Things (IoT) sustainably. Application domains of ICSs range from agricultural monitoring, supply-chain management, wild-life preservation, and smart cities all the way to monitoring space satellites.

ICSs eliminate maintenance costs (e.g., battery recharging or replacement) and thus can place a large number of small, often cheap devices in the field, even in difficult-to-access or unforgiving environments. Moreover, ICSs are essential to mitigating the environmental cost of IoT because they can operate with just capacitors instead of electrochemical cells; when IoT is projected to include trillions of devices [1], trillions of cells would represent an unsustainable and unacceptable level of e-waste.

Why Cyber-Physical Verification for ICS? ICSs are increasingly used both in safety-critical settings and settings where their performance is critical to social good. Thus it is important to provide programmers with tools, such as modeling languages and theorem-proving software, that let developers rigorously ensure ICSs meet critical constraints, lest humans be hurt or social goods be lost. Constraints are typically *cyber-physical*: ICSs are cyber-physical systems (CPSs) because they are computer systems where physics matters. ICSs are typically bound by physical access to energy sources and physical timing constraints. Most have physical sensors whose results are time-sensitive, and some may have actuators that change the physical environment. ICSs are CPSs, but not *just any* CPSs. Every ICS must contend with power, thus ICS deserve a specialized approach that confronts power issues while keeping CPS analyses tractable. Though past ICS languages [2], [3] handle restarts well, they take arbitrarily frequent restarts as a given, which makes CPS verification impossible: arbitrarily late systems are simply unsafe. We are the first to provide the converse: our energy-aware, CPS-aware language supports formal proofs in a theorem-prover by combining CPS proofs with proofs of a fundamental ICS property: *restart-freedom*, i.e., having enough power to avoid restarts. We are orthogonal: even a proven system benefits from the existing languages in practical deployments, in case it is deployed in an environment that violates the assumptions of the restart-freedom proof.

Our Solution. We take a language-based approach in order to reconcile the desire for an ICS-designer-friendly design with the need for precise physical specifications. A friendly design should center concepts well-known to designers, such as task-based programming and capacitor monitoring, yet no existing formal language for CPS verification builds in these concepts. Only by introducing a new language, which we name **BatFly**, can we provide optimal support for these practical features without losing logical rigor. BatFly is a *coordination language* for ICSs with task-based programs. Coordination languages *handle* timely, safe interaction between tasks, sensors, and data, but leave programmers free to implement tasks in any language. To ensure that an ICS program is fully safe, including its physical safety (Definition IV.1), BatFly requires physical awareness including: how often sensors produce data, how quickly the physical world changes, and how physical changes affect safety [4]. The goals of BatFly are to: (1) model ICS programs, their environments, and their assumptions using established task-based models and physical models, (2) verify that ICS programs satisfy their safety-critical constraints, under appropriate assumptions, and (3) do so for practical ICS features like capacitor arrays and variable-frequency sensing.

Correctness constraints can be written in *linear-temporal logic* (LTL) [5] to specify correctness properties that must hold over time. The LTL condition is systematically translated into a property of a **hybrid game** [6]: the ICS' control code competes against its environment to make the condition true. Hybrid games combine discrete dynamics, physical dynamics expressed as ordinary differential equations (ODEs), and adversarial competition, enabling a wide variety of properties about ICSs. Specifically, we translate the LTL specifications into *constructive differential game logic* (CdGL) [7, Ch. 5], after which existing proof tools may be used [7, Ch. 7]. CdGL notably emphasizes a similarity between proof and code, simplifying extraction of code [7, Ch. 8] from proofs. Yet CdGL is much lower-level than BatFly and even simple BatFly models would require complex models in CdGL (see Figure 9). It is thus essential to provide

Manuscript received April 07, 2022; revised June 11, 2022; accepted July 05, 2022. This article was presented at the International Conference on Embedded Software (EMSOFT) 2022 and appeared as part of the ESWEEK-TCAD special issue.

R. Bohrer and B. Islam are with Worcester Polytechnic Institute.

BatFly for greater usability than CdGL.

The core insight of BatFly proofs in CdGL is that they reduce CPS verification of ICSs into the sum of a CPS correctness proof and a restart-freedom proof. CPS correctness proof techniques are established, but restart-freedom proofs are novel. We provide the first restart-freedom proof technique, a generic technique we call *amortized energy analysis*. We prove that for each discharge, the system charges long enough to store enough energy in advance, without restarts. This technique is unique to ICS; only ICSs indefinitely alternate energy-harvesting and energy-consumption.

Application. In this paper, we pick only one of the many example use-cases of ICSs to provide an illustrative example. We draw special attention to our use of cold-chain equipment monitoring (CEM) as an example. While we are not the first to use this example [3], it is a timely, strong motivating example due to the COVID-19 pandemic. Some of the most prominent COVID-19 vaccines [8] require refrigeration, making them part of the cold-chain where correct temperature monitoring is critical to preserve vaccine effectiveness. Noticing temperature violation within bounded time is important, to allow intervention to prevent vaccine spoilage. In contrast to prior work [3], we model time and physics realistically with exponential heating and charging laws. When combined with games, these dynamics are at the cutting edge [9] of CPS theorem-proving.

ICSs are a perfect candidate: they allow cheap implementation without power grid access. Cheap implementation is crucial to provide fast and adequate vaccine supply under monetary constraints, e.g. for countries victimized by vaccine apartheid [10]. Eliminating the power grid increases the odds of providing cold-chain access to remote communities which have often been under-vaccinated due to lack or cost of power-supply to maintain the cold-chain [11]. Though COVID vaccines are a natural motivating example, ICSs' impact is broad and will continue to grow long after the pandemic.

Contributions. Following are the key contributions of this paper.

- We introduce BatFly, a CPS-centric coordination language which models the ICS program's cyber components using tasks and its physical components using ODEs, controllable by either player of a game. BatFly significantly simplifies models compared to CdGL and enables a novel task-based proof approach.
- We provide the core design insight that ICS can be verified using traditional CPS techniques, because the optimal strategy is always one that prevents power loss.
- We define a formal semantics for BatFly.
- We define a translation from LTL properties of BatFly programs into CdGL formulas about hybrid games.
- We prove the translation sound: if the CdGL formula is proved, the BatFly program is correct. To our knowledge, our refinement-based soundness approach is also novel.
- We implement the BatFly parser and translation.
- We introduce a generalizable proof paradigm for ICS correctness: amortized analysis proves a system always stores enough energy to meet later needs.
- We apply BatFly to vaccine temperature monitoring, with *the first* formal hybrid game proof that vaccines do not spoil.

The structure of the paper is as follows: Section II gives an overview of the approach and provides preliminary material in order to remain self-contained, Section III provides the BatFly syntax and example model, Section IV describes LTL for BatFly, Section V defines the meaning of LTL formulas and BatFly programs, Section VI translates BatFly to CdGL, Section VII proves the example in KeYmaera X, Section VIII proves the link between BatFly and CdGL, Section IX discusses related work, Section X discusses limitations and future work, and Section XI concludes.

II. Overview of BatFly and Preliminaries

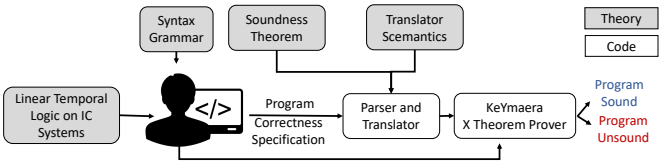

Fig. 1. Overview of BatFly.

A. Overview of BatFly

Figure 1 shows the workflow for verifying a program with BatFly. After implementing their code in a task-based model, using any language they want, an ICS developer writes a high-level program, including desired correctness properties using the syntax defined in Section III and LTL for ICS (Section IV). Then using the translation semantics of Section VI, we automatically translate the high-level program into a hybrid game and translate the correctness properties into formulas. This translation produces a theorem statement in CdGL. We prove a soundness theorem to show the ICS program meets its LTL specification whenever the CdGL translation has a KeYmaera X proof (Section VIII). Using an existing interactive proof tool KeYmaera X [12], we verify the hybrid game, then conclude that the ICS program satisfies its correctness specification.

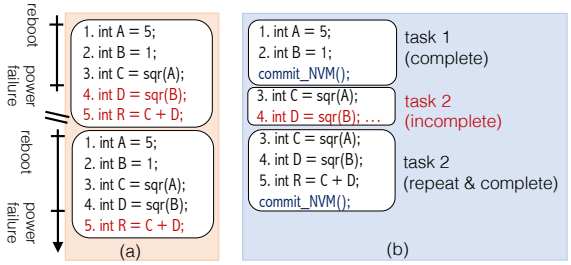

Fig. 2. Task-based models for intermittent computing.

B. Task-based Intermittent Computing

A program that executes in a persistently powered systems fails to execute correctly in an intermittent system due to power failure. In a traditional computer, all program counters and variables get reset during reboot and the program execution restarts from the beginning. When the power is intermittent, such behavior leads to failure in program execution (Figure 2a). To address this, checkpointing [2] and task-based models [3], [13] have been introduced. In this paper, we focus on task-based models where a program is divided into atomic tasks (Figure 2b). After execution of each task, the program counters and variables are saved to non-volatile memory to allow forward execution across power failures. Treatment of non-volatile memory is beyond the scope of this paper.

C. Hybrid Games

We describe hybrid games and an existing formal logic for them called CdGL, short for *Constructive Differential Game Logic*. Here, hybrid games are two-player, perfect-information, zero-sum games and players' moves use no randomness. The first player, Angel, is controlled by us while we have no control on the second player, Demon. We use the adjective *Angelic* for anything "controlled by us" and the adjective *Demonic* for anything "not controlled by us". We use standard programming language-theoretic notation for defining syntax: we write $x ::= y_1 \mid \cdots \mid y_n$ to say that an x can be n different things: any of the y's. The syntax includes terms, hybrid

games, and formulas. Hybrid games and formulas can recursively contain one another and both can contain terms, but terms contain neither formulas nor games.

Terms (variable names: e and $\tilde{e}$) are real-valued polynomials.

Hybrid Games (variable names: α and β) can be any of:

$$\alpha, \beta ::= x{:=}e \mid x{:=}* \mid ?P \mid \{x' = e \& Q\} \mid \alpha;\beta \mid \alpha \cup \beta \mid \alpha^* \mid \alpha^d$$

Deterministic assignments $x{:=}e$ assign the value of term e to x, while nondeterministic assignments $x{:=}*$ assign any value to x based on the player's choice. Tests $?P$ test whether formula P is currently true; the player loses immediately if not.

An ordinary differential equation (ODE) such as $\{x' = e \& Q\}$ changes the value of x continuously by following the evolution of the differential equation $\{x' = e\}$. The player can choose any nonnegative real number d as the duration of the ODE, so long as the *evolution domain constraint* formula Q is true at every time from 0 to d. To model an ODE with unconstrained duration, the evolution domain constraint may be omitted (equivalently to choosing $Q \equiv (1 = 1)$), written $\{x' = e\}$. ODE games support *systems* of differential equations; we merely wrote the case of a single equation for readability. Sequential composition $\alpha;\beta$ runs α, then β from any resulting state. Choice $\alpha \cup \beta$ plays either α or β depending on the player's choice. Repetition α^* repeats α any finite number of times in a loop, as chosen by the player, including 0 times.

Duality α^d plays α but switches which player is in control, e.g., Angel plays α^d by giving Demon a turn in α. For convenience, several definable game constructs are often used. Demonic repetitions $\alpha^\times \equiv \{\{\alpha^d\}^*\}^d$ mean the opposite player controls the duration of the repetition, but the current player retains control of the loop body α. Demonic choices $\alpha \cap \beta \equiv \{\alpha^d \cup \beta^d\}^d$ likewise mean the opposite player chooses whether to play α vs. play β, but the current player controls the body, respectively α or β. The other game constructs traditionally have no special Demonic syntax, as the duality operator α^d can be used to express Demonic tests, nondeterministic assignments, and ODEs without notational difficulty.

Formulas (variable names P and Q, sometimes ϕ) can be any of:

$$P, Q ::= e \geq \tilde{e} \mid \neg P \mid P \wedge Q \mid P \vee Q \mid P \rightarrow Q \mid \forall x\ P \mid \exists x\ P \mid [\alpha]P \mid \langle\alpha\rangle P$$

Comparisons $e \geq \tilde{e}$ compare two terms; all standard comparison operators can be defined using $\geq$. Negation $\neg P$ is true when P is false. Conjunction $P \wedge Q$ is true when P and Q are. Disjunction $P \vee Q$ is true when P or Q are. Implication $P \rightarrow Q$ is true when P's truth would imply Q. Quantifiers $\forall x\ P$ and $\exists x\ P$ are respectively true when all or some real-number values of x make P true. The *most important* formulas in CdGL are the modalities $[\alpha]P$ and $\langle\alpha\rangle P$ because they are the only formulas which state who wins a game. Both modalities mean Angel can win α and make P true at the end, respectively starting from Demon's ($[\alpha]P$) or Angel's ($\langle\alpha\rangle P$) turn.

A formula P is called *first-order* if it contains no modalities $[\alpha]Q$ nor $\langle\alpha\rangle Q$. When one proves a CdGL formula to be true, they prove its truth in *every* state, also called *validity*.

Definition II.1 (Validity in CdGL)**.** *A CdGL formula P is* valid*, written $\models_{\text{CdGL}} P$, if it is true in every program state ω, where a program state ω consists of a real-number value $\omega(x)$ for every variable x [7, Ch. 5].*

III. Syntax

We define the syntax for BatFly. First, we present an example scenario of COVID vaccine temperature monitoring, which we will use for the rest of the paper. Next, we define the concrete syntax, with which the user writes a program, in Section III-B. Then we define the abstract syntax, an abstraction of the concrete syntax which is more convenient for proofs, in Section III-C.

Fig. 3. Example COVID vaccine temperature monitoring task.

A. Example Scenario

We describe the example COVID vaccine temperature monitor scenario from Figure 3. The model consists of four tasks which represent four states of operation: `slow` indicates slowly sensing the temperature, `fast` means quickly sensing the temperature, `alert` means a temperature alert has been sent, and `spoiled` means the vaccines have already spoiled due to excessive heat. We use the `alert` state to model a situation where the vaccine's lid has been left open and a human must be alerted to close it.

B. Concrete Syntax

A user writes a program in BatFly by writing text that follows the syntax for SProg. The syntax for SProg along with all of its helper definitions is provided in Figure 4, but for a gentler introduction we refer the reader to our example model in Figure 5. Note that keywords are written in `monospace` font and are conceptually distinct from syntactic helper definitions, which are written in sans-serif font and are prefixed with the letter S. Optional syntax is explicitly indicated by adding an alternative ⟨*nothing*⟩ and repeated syntax is explicitly indicated (e.g. SProg SProg for repeated programs). A program SProg consists of a series of *declarations*. Typical programs SProg contain many Stask declarations, which are the heart of a program. There is also typically one Stheorem declaration to state one main theorem, one Scaps declaration to specify a capacitor array, and one SinitialValues declaration to specify the initial values of every variable. In Figure 4, n means any numeric literal, *id* means any literal identifier, and P is a first-order formula (see Section II-C).

SProg ::= Stask | Stheorem | Scaps
| SinitialValues | SProg SProg
Stask ::= Smodifier `task` *id* {SresetC Sode Sjumps}
Smodifier ::= `angel` | `demon` | ⟨*nothing*⟩
SresetC ::= `set` {Sresets} | ⟨*nothing*⟩
Sresets ::= $x := e$ | $x := *$ | $?P$ | Sresets Sresets
Sode ::= `ode`{Sequations} | ⟨*nothing*⟩
Sequations ::= $d\_x = e$ | Sequations, Sequations
Sjumps ::= `jumps before` P {Sedg} | `jumps` Sedg
| ⟨*nothing*⟩
Sedg ::= {`to` *id* SwhenC SresetC} | Sedg Sedg
SwhenC ::= `when` P | ⟨*nothing*⟩
Stheorem ::= `theorem` P
Scaps ::= `capacitors` { Snumbers }
SinitialValues ::= `initial_values` { Svalues }
Svalues ::= *id* $=$ n | Svalues Svalues
Snumbers ::= n | Snumbers Snumbers

Fig. 4. Concrete syntax

The bulk of a program is a series of Stask declarations. By convention, the first task to run is the one listed first. Each task optionally starts with a Smodifier indicating which player controls the task's ODE duration and resets. The players Angel and Demon are respectively under and not under our control. The default is Demon, as Demonic control is a conservative modeling assumption. The optional modifier is followed by the keyword `task`, then the task name (an *id*entifier), then the task body in braces. The body consists of a reset (SresetC), ODE (Sode), and jumps (Sjumps), each of which are optional, so an empty task body is valid syntax.

The optional SresetC starts with the `set` keyword, which is followed by Sresets in braces. Each reset is either a deterministic assignment of a variable to the current value of a term ($x := e$), a nondeterministic assignment in which the current player, Angel or Demon, picks a new numeric value of x, written $x := *$, or a guard $?P$ which must hold true for the player's new chosen values.

An ordinary differential equation (ODE) declaration starts with the `ode` keyword followed by equations in braces. There is one equation for each variable that changes continuously in the given task. If there are multiple equations, they are separated by commas. We write $d\_x = e$ to indicate that at each instantaneous moment, the derivative of x with respect to time is equal to the current value of the real-valued polynomial term e. Typically, every ODE system includes an equation which describes the rate at which the capacitor charges or discharges. For example, d_capacitor $= 1$ means the net charging rate is 1V per unit time and d_capacitor $= -1$ means discharging at the same rate. As always in hybrid systems, time is continuous rather than discrete, i.e., at *every* instant in time, a variable's rate of change obeys the ODE. The variable capacitor is a special built-in variable which always stands for whichever capacitor is currently in use, likewise `capacitor_max` for its maximum voltage. By convention, the first capacitor listed in the Scaps declaration is "in use" at the beginning of the program.

The Sjumps declaration starts with the `jumps` keyword, optionally followed by the keyword `before` and a formula P, before proceeding to each jump (Sedg) in braces either way. If provided, the `before` clause indicates that a jump *must* occur *before* the formula P becomes true, thus limiting from above the duration spent in the task. This is not to be confused with the SwhenC clause of each jump, which allows bounding duration from *below*. Each jump has the keyword `to` followed by the name of the destination task (i.e., the task we transition to), an optional SwhenC and an optional SresetC. The SwhenC, if provided, contains the `when` keyword followed by a formula P, meaning that the transition can only occur when P is true. It is common to have a `SwhenC` for every jump and also a shared `before` condition. In this case, at least one `SwhenC` condition must become true before the `before` condition becomes true in order for execution to continue to the next task.

Note that a SresetC can appear both at the beginning of a task body and within a specific jump. In the former case, the reset is always executed when the task starts. In the latter case, the reset is executed only when the given jump is taken, in which case it executes at the end of the task. Note that the abstract syntax of Section III-C only allows jumps during resets, because the former class of jumps can be readily translated into the latter [1].

A Stheorem declaration begins with the keyword `theorem`, which is followed by a formula, according to the LTL formula syntax to be defined in Section IV.

A Scaps declaration begins with the keyword `capacitors` followed by a list of numbers, in braces. There are three numbers for each capacitor: minimum voltage, maximum voltage, and charging coefficient, in that order. For example, `1.8 7.0 0.1` indicates a capacitor that needs 1.8V to run, can hold 7.0V, and charges at rate $0.1 \cdot (7 - x)$ when in recharging mode, where x is current voltage.

The SinitialValues declaration begins with the keyword `initial_values`, followed by assignments of initial values for each variable, in braces. Numeric initial values are specified for all variables written or read by the program.

```
angelic task slow {
 set {stopwatch := 0;}
 ode {
  d_stopwatch = 1, d_clock = 1,
  d_capacitor = 0.1*(capacitor_max-capacitor),
  d_temp = 0.0005*(23-temp)}
 jumps before clock = 60*24 {
  {to fast when
    stopwatch >= 60 & clock < 12*60+1}
  {to alert when
    stopwatch >= 60 & clock >= 12*60+1}}}
angelic task fast {
 set {stopwatch := 0;}
 ode {
  d_stopwatch = 1, d_clock = 1,
  d_capacitor = -4,
  d_temp = 0.0005*(23-temp)}
 jumps before clock = 60*24 {
  {to slow when
    stopwatch >= 1 & clock < 12*60+1}
  {to alert when
    stopwatch >= 1 & clock >= 12*60+1}}}
demonic task alert {
 set {stopwatch := 0;}
 ode {
  d_stopwatch = 1, d_clock = 1,
  d_capacitor = -1, d_temp = 0.09*(23-temp)}
 jumps before stopwatch > 3 {
  {to slow set {
    temp := *;
    ?(-90 <= temp & temp <= -60);
    clock := 0;}}
  {to spoiled when temp >= 8}}}
demonic task spoiled {}
capacitors {1.8 7 0.1 1.8 7 0.1}
initial_values { temp = -60 stopwatch = 0
    clock = 0 capacitor1 = 2 capacitor2 = 2}
theorem □ (!{spoiled} & temp < 8)
```

Fig. 5. Example program

Figure 5 shows the syntax of the example program described in Section III-A. Here, `slow` and `fast` are Angelic tasks while `alert` and `spoiled` are Demonic tasks where Demon is in control. The `spoiled` task is empty because no code needs to be run; this state simply indicates failure. The other tasks all use a local `stopwatch` that resets on each transition, a global `clock` that resets only after alerts, and a `capacitor` which changes at different rates: slow sensing requires energy, so it is the only state that accumulates energy, while the others discharge. Each state also models changes in temperature `temp`, with the `alert` state having the highest rate because we assume the lid may be open. In Angelic tasks, Angel controls ODE duration, subject to the time limit from the evolution domain constraint. Angelic control is realistic here because Angel has the control to switch tasks or yield execution; it is also needed

[1] A start-of-task reset is implemented by appending it to the reset for every jump *into* that task.

since it lets Angel distributing charging across capacitors.

The `jumps` indicate the `slow` and `fast` states may switch among themselves only when their control code runs after each sensing cycle [14]. Because they should only have to `alert` when a lid is left open and the lid is supposed to be opened infrequently [8], we can assume a long interval (`clock`) elapses before any jump to `alert`, crucially meaning the `slow` state has time to recharge capacitors before the `fast` and `alert` states discharge them. The `alert` task `jumps` back to `slow` if 3 time units elapse; we assume this is how long it takes a human to come close the lid. At that time, we assume the human adds coolant, which crucially lowers the temperature. We give Demon significant control over the new temperature, from -90 to -60 degrees. Otherwise if temperature reaches 8 degrees Celsius before the lid closes, the vaccine spoils, represented by the `spoiled` state. The exact `capacitors` declaration is inessential, but we importantly use two capacitors so that one may be dedicated to quickly handling the `alert` task whenever needed. The `initial_values` are set to a near-worst-case: we just exited an alert at the highest possible temperature with low energy. This is important because when verifying a system's correctness, we need to prove it operates correctly even in the worst case.

The theorem statement of Figure 4 uses LTL notation (see Section IV). The notation $\Box P$ means property P must hold at all times. Thus, the theorem statement says that the system never enters the spoiled state and that its temperature never exceeds 8 degrees Celsius. We automatically translated this system and theorem to CdGL with (our implementation of) the translation we define in Section VI, then formally proved that the system satisfies the theorem in KeYmaera X (VII). We use realistic models of capacitors and temperature. The system heats exponentially toward the ambient temperature (e.g., `d_temp = 0.0005*(23-temp)`). Capacitor charging follows a standard exponential charging law (e.g., `d_capacitor = 0.1*(capacitor_max-capacitor)`). Realistic capacitor *dis*charging, however, is linear, because power is consumed at a constant rate. The upper and lower voltage limits of 1.8V and 7V are standard in the literature [15].

C. Abstract Syntax

We present a formal grammar for the abstract syntax of BatFly, which will be more convenient for the theoretical developments of Section V and Section VI, and Section VIII compared to concrete syntax. A program in BatFly is organized as a task graph G. The whole task graph represents one single-threaded program in which the individual tasks divide the program into pieces. As is typical in task-based ICS languages, programs resume at the start of a task when a system restarts, so the division of a program into tasks serves to inform the management of restarts. The program runs in a cyber-physical environment with some set of global variables $x \in \mathcal{V}$ where $\mathcal{V}$ means the set of all variable names. For modeling purposes, the value of each variable x is a real number, though an implementation might approximate real numbers with floating-point or fixed-point numbers. The variables are globally scoped and shared between tasks. Variables can be changed both by physics and code.

A task graph G consists of a vertex set and edge set, written $G = (V, E)$. Each vertex represents a task and each edge represents a possible transition between tasks (u or v). Tasks $u = (n, f)$ have names n and are labeled with a system of ordinary differential equations (ODEs) f (also called a *plant*) describing continuous physical changes during that task. In BatFly, every ODE f must be locally Lipschitz continuous, a very modest assumption which ensures the ODE has a unique solution [16, §10.VII] and thus simplifies its analysis. Optionally, f can include an *evolution domain constraint* P; f may evolve only while P remains true. For example, $\{x' = 1, y' = x \& x \leq 10\}$, evolves only while $x \leq 10$ holds, with $x' = 1$ and $y' = x$ defining the time derivatives of x and y at each instant. When an ODE can evolve no further, G typically performs a discrete transition next. Throughout the paper, guards P are first-order (Section II-C) with no quantifiers.

Every task $u = (n, f)$ is controlled by exactly one of the two players Angel or Demon, respectively written $u \in \mathsf{VA}$ or $u \in \mathsf{VD}$. The player in control of the task gets to control any nondeterministic choices made during its execution, i.e., the amount of time spent in the task and potentially which task is executed next. Edges are labeled with guard formulas ϕ describing when that transition is allowed to occur and discrete resets D (modeled as a relation, equivalently a 1-player game) describing how the state changes immediately on entering the task, so that an edge is written (u, v, ϕ, D). Each pair (u, v) has at most one edge $(u, v, \phi, D) \in E$ because G is a graph.

A reset is a set of assignment statements which execute sequentially and define new values of state variables, potentially as a function of old ones. If we wish to leave the new value unspecified (i.e., allow it to take any value the player chooses), we use the notation $x{:=}*$. For example, the reset $x{:=}2; y{:=}*; z{:=}x + y$; sets x to 2, y to any value, and z to the sum of the *new* values of x and y.

As a coordination language, it is by design that we abstract away many of the low-level implementation details of a full program. Nondeterministic assignments $z{:=}*$ are one of the main tools for that abstraction: if a task computes a variable z but we do not know nor care how exactly z is computed, then we can specify that its value is arbitrary by writing $z{:=}*$.

We have not specified anything about the language used to implement individual tasks. This is intentional, because BatFly is intended to be used with the implementation language of your choice.

IV. Linear Temporal Logic on ICS Systems

We have defined BatFly programs, but still need a way to state correctness properties about them. Two of the most common classes of correctness properties about programs are properties that hold at *all* times or *some* time. The former can show programs always avoid some undesirable state, while the latter can show they eventually reach a desired state. Linear Temporal Logic (LTL) [5] is a widely used logic whose operators $\Box P$ and $\Diamond P$ respectively capture this notion that formula P holds at *all* or *some* times. We place special emphasis on formulas $\Box P$ used for (physical) safety properties.

Definition IV.1 (Safety and Physical Safety)**.** *A safety is any property of form $\Box P$, i.e., a safety property $\Box P$ is defined as the property that some safety condition P holds at all times. A* physical safety *property is any property $\Box P$ where P mentions variables that describe physical state.*

For example, the physical safety property $\Box \mathit{temp} \leq 8$ means temperature never exceeds 8 degrees (Section III-A). This notion generalizes to any CPS. For example, physical safety properties for cars include $\Box \mathit{speed} \leq \mathit{speedLimit}$ and $\Box \mathit{car1Pos} \neq \mathit{car2Pos}$.

There are many variants of LTL which often add their own formulas P. We add to P task specifications S which are sets of task names $S = \{n_1, \ldots, n_k\}$. Formula S is true if the current task name is any of the n_k. For convenience, we also assume that P includes first-order logic, though LTL does *not* include CdGL.

Because our ICS programs are games, our LTL over ICS programs is a temporal logic over games. Ours is far from the first such logic [17], but such logics remain noteworthy theoretical contributions even today. We highlight the subtle interaction between time and games. Where Angel is the game's first player, our $\Box P$ means Angel has a strategy to make P true at all times for all Demon strategies

and our $\Diamond P$ means Angel has a strategy to make P true during at least one moment, for all Demon strategies. Counterparts for Demon could be defined, but Angelic versions are typically most useful for proving program correctness, as discussed in the literature [7, Ch. 5].

We will use *trace semantics* (Section V) to describe a system's behavior over time. Thanks to the generality of trace semantics, most LTL operators can be combined freely (e.g. $\Box(\Diamond P \rightarrow \Box Q)$) because the meanings of most formulas (called *trace formulas*) are well-understood for arbitrary traces. However, the behavior of trace formulas inside quantifiers is deemed confusing, so quantifiers $\forall x\ P$ and $\exists x\ P$ require modality-free formulas P, known as *state formulas*. Propositional connectives can appear in both state and trace formulas.

V. Trace Semantics

It is critical to define what programs and formulas mean, otherwise there is no way to rigorously show whether a program is correct, nor define what it would mean to be correct. The formal mathematical meaning of a program or formula is called its *semantics*. There are multiple well-established approaches for defining semantics. Among those, we define formulas' and programs' primary meanings using *trace semantics*, which defines the meaning of a program as a *trace* listing out its sequential behavior at subsequent moments in execution. We use it because it excels at addressing our LTL formulas which must always be true (every moment in the trace) or must eventually be true (there exists a moment in the trace). In Section VI, we translate our LTL formulas to CdGL formulas to exploit CdGL proof tools. Crucially, Section VIII will link the LTL trace semantics to the CdGL translation: if the CdGL translation has a proof, the LTL formula is true according to the trace semantics.

We now define one of the key concepts for the semantics: how to represent a moment in the execution of a program. We represent each moment as a *machine configuration* $m = (n, M)$, which contains the name n of the current task and a map M representing the current values of state variables. The map M assigns a real-number value $M(x)$ to every non-volatile variable x. We assume that all non-volatile variables are stored to non-volatile memory upon every task transition. Our semantics does not explicitly model volatile memory; if one implements a system modeled verified in BatFly, we leave this important task to the implementer. In addition to the non-volatile variables, we model several built-in variables to track the current energy stored in every capacitor. We write nQ for the number of capacitors, then for every $i \in [1, \mathsf{nQ}]$, the variable q_i refers to the current energy stored in capacitor i. We assume we have been given constants $Qmax_i, Qmin_i$, and qrate_i which stand for the minimum voltage, maximum voltage, and recharging coefficient of each capacitor i. A variable qp indicates which capacitor i is currently being used. Being constants, they need not be stored as variables and thus do not appear in the map. Note that energy consumption occurs during every task, often at different rates, so the derivative of q_{qp} typically differs across different states. Typically, the derivative of q_{qp} is negative in some states and positive than others, depending on whether energy use is greater or less than energy harvested. As a result, if the system spends enough of its time in energy-increasing states, it need never run out of energy. A moment m describes only one instant in time. To reason about physics evolving over time, we write $\varphi \models_{[0,t]} f$ to mean the solution function φ solves the differential equation f on the time interval $[0, t]$. If f contains an evolution domain constraint, $\varphi \models_{[0,t]} f$ also means the domain constraint is true everywhere on $[0, t]$. We write $M[x \mapsto v]$ to mean a map where the value of variable x is updated to v but all others are unchanged from M.

The above notations suffice to describe the current state of a program in BatFly, but running that program also requires calling implementation code. Thus the semantics must mention task implementations for Angel and Demon, respectively written $AI()$ and $DI()$. An implementation decides which transition to make next. Because transitions can be either discrete transitions, continuous transitions, or switching between capacitors, we write $AI((u, M)) = Disc((v, M'))$ when the implementation tells us to make a discrete transition to machine configuration (v, M'), we write $AI((u, M)) = Cont(t)$ when the implementation tells us to make a continuous transition of duration t, and we write $Cap(i)$ when the implementation tells us to switch to capacitor i. Instead of transitioning, the implementation can also tell us that execution ends, written $AI((u, M)) = Stop$. The notations for Demon are likewise. Not every function is suitable as an implementation, but if so, we call the function *admissible* (Definition V.1).

Definition V.1 (Admissible implementations)**.** *An Angelic implementation $AI()$ (and likewise for Demonic implementations $DI()$) is* admissible *if for all machines (u, M):*

1) *Either $AI((u, M)) = Disc((v, M'))$ for some v, M' or $AI((u, M) = Cont(t))$ for some real number $t \geq 0$, $AI((v, M')) = Cap(i)$ for some $i \in [1, \mathsf{nQ}]$, or $AI((u, M)) = Stop$.*
2) *Whenever $AI((u, M)) = Disc((v, M'))$, there exist ϕ and D such that $(u, v, \phi, D) \in E$.*
3) *Whenever $AI((u, M)) = Disc((v, M'))$, then $(M, M') \in D$.*
4) *Whenever $AI((u, M)) = t$, let $(n, f) = u$, then have $\varphi \models_{[0,t]} f$ where φ is the unique solution of f such that $\varphi(0) = M$.*
5) *If no transition rule applies, $AI((u, M)) = Stop$ must hold.*

Requirement 1 means that the implementation must always return *some* decision. Requirement 2 means that every discrete transition must follow some edge from the task graph. Requirement 3 intuitively means $AI((u, M))$ implements the reset specification D. Often, D is an *abstract* specification for $AI((u, M))$, e.g., it may leave some variables unspecified. Requirement 4 means f has a solution that exists (and satisfies any evolution domain constraint) for at least time t . Requirement 5 says a program must end if it cannot continue.

We are now finally ready to define the semantics of programs. Formally, we write $[\![G]\!]$ for the semantics (meaning) of program G. Program semantics are defined by simultaneous induction with formula semantics $[\![P]\!]_G$, meaning each definition can refer to the other, though we present the definition for programs first. The semantics $[\![G]\!]$ of program G is the set of all execution traces of G. Each trace is written in full as a comma-separated list of machine configurations, or abbreviated with vector notation $\vec{m}$. For example, we write $m_1, \ldots, m_n \in [\![G]\!]$ (or $\vec{m} \in [\![G]\!]$ for short) to indicate that some program trace starting with configuration m_1 and ending with m_n can occur when running G from initial configuration m_1. A trace is typically non-empty but can certainly be a singleton: $m \in [\![G]\!]$ represents the base case where we have chosen an initial configuration m but have not actually executed any code, thus no changes to the configuration have occurred. The semantics $[\![G]\!]$ captures both traces where programs run to completion and also intermediate traces of running programs. We write $\mathsf{ends}(m_1, \ldots, m_n)$ for traces that run to completion. Letting $m_n = (u, M)$ then $\mathsf{ends}(m_1, \ldots, m_n)$ holds iff $u \in \mathsf{VA}$ and $AI((u, M)) = Stop$ or $u \in \mathsf{VD}$ and $DI((u, M)) = Stop$. The full trace semantics are defined in Figure 6. In each case let $(V, E) = G$ and $(n, f) = u$.

Equation (1) is the base case, which indicates we may choose to start execution from any configuration m. Equation (2) is how larger execution traces are constructed from individual steps; it allows two traces to be concatenated so long as the final configuration of the first is the initial configuration of the second. Equation (3) allows

$$m \in [\![G]\!] \quad (1)$$

$$\vec{m}_1, m, \vec{m}_2 \in [\![G]\!] \text{ if } \vec{m}_1, m \in [\![G]\!], m, \vec{m}_2 \in [\![G]\!] \text{ and not } \mathsf{ends}(\vec{m}_1, m) \quad (2)$$

$$(u,M),(u,M') \in [\![G]\!] \text{ if } AI((u,M))=Cap(i), M'=M[\mathsf{qp} \mapsto i] \quad (3)$$

$$(u,M),(v,M') \in [\![G]\!] \text{ if } u \in \mathsf{VD}, DI((u,M))=Disc((v,M')) \text{ and exists } \phi, D \text{ s. t. } (u,v,\phi,D) \in E \text{ and } (u,M) \in [\![\phi]\!]_G \text{ and } (v,M') \in [\![\forall i\ q_i \geq \mathit{Qmin}_i]\!]_G \quad (4)$$

$$(u,M),(v,M') \in [\![G]\!] \text{ if } u \in \mathsf{VD}, DI((u,M)) = Cont(t) \text{ and exists } \varphi \models_{[0,t]} f \text{ s. t. } \varphi(0) = M \text{ and } \varphi(t) = M' \text{ and } (v,M') \in [\![\forall i\ q_i \geq \mathit{Qmin}_i]\!]_G \quad (5)$$

$$(u,M),(v,M') \in [\![G]\!] \text{ if } u \in \mathsf{VA}, AI((u,M))=Disc((v,M')) \text{ and exists } \phi, D \text{ s. t. } (u,v,\phi,D) \in E \text{ and } (u,M) \in [\![\phi]\!]_G \text{ and } (v,M') \in [\![\forall i\ q_i \geq \mathit{Qmin}_i]\!]_G \quad (6)$$

$$(u,M),(v,M') \in [\![G]\!] \text{ if } u \in \mathsf{VA}, AI((u,M)) = Cont(t) \text{ and exists } \varphi \models_{[0,t]} f \text{ s. t. } \varphi(0) = M \text{ and } \varphi(t) = M' \text{ and } (v,M') \in [\![\forall i\ q_i \geq \mathit{Qmin}_i]\!]_G \quad (7)$$

$$(u,M),(u,M') \in [\![G]\!] \text{ if } M(q_{\mathsf{qp}})=\mathit{Qmin}_i \text{ and exists } t \geq 0 \text{ s. t. } \varphi \models_{[0,t]} \hat{f} \text{ and } \mathit{dq} \leq 0 \text{ and } \varphi(0) = M \text{ and } \varphi(t) = M' \text{ (let } (g,q'_{\mathsf{qp}} = \mathit{dq})=f, \hat{f}=(g,q'_{\mathsf{qp}} = \mathsf{qrate}_i)) \quad (8)$$

$$(u,M),(u,M'') \in [\![G]\!] \text{ if } u \in \mathsf{VD}, DI((u,M))=Disc((v,M')) \text{ and exists } \phi, D \text{ s. t. } (u,v,\phi,D) \in E \text{ and } (u,M) \in [\![\phi]\!]_G \text{ and } (u,M') \notin [\![\forall i\ q_i \geq \mathit{Qmin}_i]\!]_G \text{ and } M'' = M[q_{\mathsf{qp}} \mapsto 0] \quad (9)$$

$$(u,M),(u,M'') \in [\![G]\!] \text{ if } u \in \mathsf{VA}, AI((u,M))=Disc((v,M')) \text{ and exists } \phi, D \text{ s. t. } (u,v,\phi,D) \in E, (u,M) \in [\![\phi]\!]_G, (u,M') \notin [\![\forall i\ q_i \geq \mathit{Qmin}_i]\!]_G, \text{and } M'' = M[q_{\mathsf{qp}} \mapsto 0] \quad (10)$$

Fig. 6. Trace semantics

switching to a different capacitor. Note that capacitor switching is *always* controlled by Angel, even at Demonic tasks.

The next four cases implement individual, successful execution steps, i.e., individual discrete and continuous transitions. There are four such rules: a task can be Angelic vs. Demonic, and a transition can be discrete vs. continuous.

Equation (4) implements a successful discrete transition (i.e., switching to a new task) from a Demonic task. Specifically: if we are running a Demonic task ($u \in \mathsf{VD}$) then we consult Demon's implementation $DI((u,M))$ and ask it which transition to make. If the implementation tells us to attempt a discrete transition ($Disc((v,M'))$) then we look up the edge associated to the transition ($(u,v,\phi,D) \in E$). This rule applies if the transition succeeds, i.e., when the guard holds ($(u,M) \in [\![\phi]\!]_G$) the active capacitor contains enough energy to complete the transition ($(v,M') \in [\![\forall i\ q_i \geq \mathit{Qmin}_i]\!]_G$). Demon's implementation is responsible for tracking energy consumption and also implementing the reset logic D. Note that although different rules are used for transitioning *from* and Angelic task and transition *from* a Demonic task, those rules work regardless whether the *destination* task is Angelic vs. Demonic.

Equation (5) implements a successful continuous transition from a Demonic task. When the Demonic implementation tells us to attempt a continuous transition ($Cont(t)$), we do so, specifically we allow the current continuous dynamics f to evolve for time $t \geq 0$. Recall that $\varphi(0) = M$ and $\varphi \models_{[0,t]} f$ mean φ is the unique solution of the initial value problem $\varphi(0) = M$ and $\varphi(t)' = f$ on interval $[0,t]$. Recall that f can contain an *evolution domain constraint* which must be true at all times the task is running; implicitly, the domain of φ is a prefix of $[0,\infty)$ such that the evolution domain constraint holds everywhere on the domain. The transition only succeeds if enough energy remains by time t ($(v,M') \in [\![\forall i\ q_i \geq \mathit{Qmin}_i]\!]_G$). Note that depending on the task, the capacitor may be charging, in which case the energy constraint is automatically satisfied, or discharging, in which case satisfaction of the constraint is not automatic for all t.

The Angelic counterparts of the previous two rules are respectively Equation (6) and Equation (7), which differ from the Demonic rules only in that they consult the Angelic implementation ($AI((u,M))$).

The remaining three rules cover recharging after energy is exhausted and recognizing when energy is exhausted during a discrete transition (discrete failure). The failure rules for continuous transitions would be redundant with discrete failures and are thus omitted. In an ICS, these behaviors are not considered errors, because power is intermittent. When we *verify* ICSs (Section VII), however we will prove that the behaviors do not occur. That is, we will prove that an ICS harvests enough energy to avoid unplanned power loss.

Equation (8) is a built-in charging mode which can only activate when energy reaches its minimum while running a task that drains energy. That is, letting *dq* stand for the derivative of q_{qp} in task u and letting g stand for the rest of the ODE, then the mode activates if *dq* is non-positive and q_{qp} is Qmin_i, then charges at maximum rate while the physics g continue to evolve. A system is restart-free iff it never needs Equation (8).

Equation (9) describes a Demonic task which attempts a discrete transition but runs out of energy ($(u,M') \notin [\![\forall i\ q_i \geq \mathit{Qmin}_i]\!]_G$) and shuts down temporarily, until it can reboot. Though not always convenient, this is an expected behavior, not an error. In this case, the step ends in the current configuration, but with the current capacitor at minimum voltage($M'' = M[q_{\mathsf{qp}} \mapsto \mathit{Qmin}_i]$) due to power loss. Afterwards, energy will be accumulated. If the current task is one which charges energy, then the current task's continuous transition may resume right away (Equation (5)). Otherwise, the only choice is to spend time in the built-in charging mode (Equation (8)) until sufficient energy is accumulated. Thereafter, a discrete transition can be undertaken again, this time successfully (Equation (4)). Recall that we have globally assumed $Imp((u,M))$ is admissible with respect to D, which means that the transition from old to new state satisfies the guard relation D. The Angelic counterpart of Equation (9) is Equation (10). Note there is no specific failure rule for continuous transitions, though it is certainly possible for power to run out during a continuous transition. However, existing rules handle this case: a successful continuous transition ends right when energy reaches the minimum, followed by a recharge.

This completes giving ICS programs formal meaning by defining their trace semantics. Next we define the semantics of LTL *formulas*. Recall that a *state formula* has a meaning in each individual state and a *trace formula* has a meaning defined over a trace of states. We define the semantics first for state formulas (Figure 7), then trace formulas (Figure 8). The state semantics and trace semantics are standard for first-order logic and LTL, respectively. The semantics show that every state formula can be reinterpreted as a trace formula by evaluating it in the initial state of the trace. The converse is not true, however, LTL modalities can only be understood as trace formulas and cannot be reinterpreted as state formulas. For state semantics, we write $m \in [\![P]\!]$ when P is true in configuration m. State formula semantics are independent of which program is being executed; they omit G.

The trace formula semantics are defined in terms of the trace semantics of ICS programs. We write $\vec{m} \in [\![P]\!]_G$ when formula P is true of the trace $\vec{m} = m_1, \ldots, m_n$ of graph G and $m_1, \ldots, m_n \notin [\![P]\!]_G$ when it is false. Boxes and diamonds are the two cases which make fundamental use of traces: $\Box P$ is true of a trace when P is true of all nonstrict suffixes, i.e., when P is true both now and always in

$$m \in [\![P \wedge Q]\!] \quad \text{iff } m \in [\![P]\!] \text{ and } m \in [\![Q]\!] \tag{11}$$
$$m \in [\![P \vee Q]\!] \quad \text{iff } m \in [\![P]\!] \text{ or } m \in [\![Q]\!] \tag{12}$$
$$m \in [\![P \to Q]\!] \quad \text{iff } m \in [\![P]\!] \text{ implies } m \in [\![Q]\!] \tag{13}$$
$$m \in [\![\forall x\ P]\!] \quad \text{iff } m[x \mapsto r] \in [\![P]\!], \text{ all } r \in \mathbb{R} \tag{14}$$
$$m \in [\![\exists x\ P]\!] \quad \text{iff } m[x \mapsto r] \in [\![P]\!], \text{ some } r \in \mathbb{R} \tag{15}$$
$$m \in [\![\neg P]\!] \quad \text{iff } m \notin [\![P]\!] \tag{16}$$

Fig. 7. State formula semantics

$$\vec{m} \in [\![\Box P]\!]_G \quad \text{iff } m_i, \ldots, m_n \in [\![P]\!]_G, \text{ all } i \in [1, n] \tag{17}$$
$$\vec{m} \in [\![\Diamond P]\!]_G \quad \text{iff } m_i, \ldots, m_n \in [\![P]\!]_G, \text{ some } i \in [1, n] \tag{18}$$
$$\vec{m} \in [\![P \wedge Q]\!]_G \quad \text{iff } \vec{m} \in [\![P]\!]_G \text{ and } \vec{m} \in [\![Q]\!]_G \tag{19}$$
$$\vec{m} \in [\![P \vee Q]\!]_G \quad \text{iff } \vec{m} \in [\![P]\!]_G \text{ or } \vec{m} \in [\![Q]\!]_G \tag{20}$$
$$\vec{m} \in [\![P \to Q]\!]_G \quad \text{iff } \vec{m} \in [\![P]\!]_G \text{ implies } \vec{m} \in [\![Q]\!]_G \tag{21}$$
$$\vec{m} \in [\![\neg P]\!]_G \quad \text{iff } \vec{m} \notin [\![P]\!]_G \tag{22}$$
$$\vec{m} \in [\![P]\!]_G \quad \text{iff } m_1 \in [\![P]\!] \text{ for state formula } P \tag{23}$$

Fig. 8. Trace formula semantics

the future, whereas $\Diamond P$ is true when P is true of at least one nonstrict suffix, i.e., when it is true either now or some time in the future. The propositional connectives $\wedge, \vee, \to, \neg$ respectively have their standard meanings, which are the same as for CdGL in Section II-C. When the propositional connectives appear in trace formulas, their propositional meaning is applied to every configuration in the trace. Lastly, every state formula is a trace formula; its semantics is equivalent to its state formula semantics in the initial state of the trace. This semantic rule implicitly restricts us from nesting modalities inside of quantifiers: the trace semantics of quantifiers such as $\forall x\ P$ are defined using the state formula semantics of $\forall x\ P$, thus using the state semantics of P, prohibiting the use of modalities in P as described in Section IV.

The state formula and trace formula semantics tell us whether a formula is true for a certain state or trace, but we wish to verify that a formula is true *in general*, because a program should *always* satisfy its correctness specification. Thus, just as Definition II.1 defined *validity* for CdGL, we define validity for LTL using trace formula semantics. Once validity is defined, verifying a program's correctness amounts to verifying that its correctness specification is a valid formula. A correctness specification should hold for all execution traces of the program, and it should hold no matter what Demon does. However, it does not need to hold for every possible Angel behavior, because we get to choose Angel's behavior. These considerations result in the following definition of validity:

Definition V.2 (LTL Validity for BatFly)**.** *An LTL formula P is valid for a program G, written $\models_G P$, if there exists an admissible Angel implementation $AI()$ such that for all admissible Demon implementations $DI()$, all traces such that $\mathsf{ends}(m_1, \ldots, m_n)$ and $m_1, \ldots, m_n \in [\![G]\!]$ satisfy $m_1, \ldots, m_n \in [\![P]\!]$.*

This completes the definition of the semantics for formulas, and thus also completes the definition of the trace semantics overall.

VI. Translational Semantics with Hybrid Games

Though the trace semantics of Section V importantly defined which formulas should be considered true, it does not provide a means to prove truth. To that end, we now introduce a *translational* semantics: we translate LTL formulas into CdGL formulas, for which proof techniques already exist. Though the translational semantics enables important proof techniques, it was important to start with trace semantics because they are mathematically simpler, and it is equally important to show the translation is sound (VIII) so when we prove a CdGL formula, we can conclude the matching LTL formula is true. The basics of hybrid games are in Section II-C and details of CdGL are in the literature [7, Ch. 5].

Formula Translation. We translate an LTL formula P over graph G into a CdGL formula ϕ; translation is written $P \rightsquigarrow_G \phi$. The translations for $\Box P$ and $\Diamond P$ specifically involve translating task graphs into hybrid games, but $\Box P$ and $\Diamond P$ use *different* game translations from each other in order to capture the difference between them. For any task u, we write $out(u) = \{(n', f', \phi, D) \mid (u, (n', f'), \phi, D) \in E\}$ for the contents of u's outgoing edges. Additionally, the translation assumes that done stands for any fixed sentinel value which is guaranteed to be distinct from the value assigned to every task id n. That is, $\mathsf{pc} = \mathsf{done}$ indicates that no task is running. We also assume the existence of built-in variables capacitor, qp, and $Qmax_{tmp}$ which we use for the special purposes of storing the current capacitor's charge, index, and max voltage, respectively. We first explain all the auxiliary definitions used in the translation, then explain each case of the translation one-by-one.

Definitions αjmp and δjmp implement Angelic and Demonic discrete transitions, respectively. In each case, the player chooses an out-edge of the current task u, shows the guard ϕ holds, applies the reset, updates the task name, and shows the capacitor is nonempty. It is essential to show the capacitor is nonempty after the transition; this is part of how one proves that the program executes without needing restarts. If multiple guard formulas are true for different tasks, the player gets to choose which task to transition to, as indicated by the choices $\cup$ and $\cap$. Definitions getc and pickc respectively load information about the current capacitor into temporary variables and implement capacitor-switching transitions. Definition maxc caps the capacitor's charge at its limit $Qmax_{tmp}$. Charging constraint QC forbids negative capacitor charges during evolution. The next four definitions implement continuous transitions. Each one first tests that the transition can only be applied during its corresponding task, then allows physics to evolve, then caps the capacitor charge. Respectively, the definitions $\alpha\Box f, \delta\Box f, \alpha\Diamond f$, and $\delta\Diamond f$ handle Angelic and Demonic tasks in box formulas and Angelic and Demonic tasks in diamond formulas. In each definition, the corresponding player is responsible for choosing the ODE's duration. In box formulas, Angel is responsible for showing the postcondition P' at all times. At an Angelic task, that is done using a domain constraint; at a Demonic task, it is done using a test following an ODE whose duration Demon chose, (i.e., the test must hold for *every* duration). Definition $\alpha lastf$ is used only for proving diamond properties, specifically the special case where a diamond property becomes true during a Demonic task. Recall in the definitions that pc stores the currently-running task's identity, VA and VD respectively stand for the sets of Angelic and Demonic tasks, and that resets D, which are described as relations in Section III-C can equivalently be understood as one-player games, i.e., nondeterministic programs.

In the first equation, recall that formulas $\{u_1, \ldots, u_n\}$ are true exactly when one of the tasks u_i is running for some $i \in [1, n]$. Thus the translation compares the name of the current task against the name of each u_i and is true exactly when it matches one of them.

In the second equation, op stands for any first-order logic operator. This equation simply means that whenever we encounter these operators, we just translate the operands recursively. For example, when encountering a conjunction $P \wedge Q$, we translate each of P and Q, then conjoin the results. The second equation implies as a special case that first-order formulas translate to themselves.

The translations for $\Box P$ and $\Diamond P$ are the most involved. In each case, let P' be the translation of P. Then $\Box P$, which means P

$$\alpha\mathsf{jmp} \equiv \cup_{(n',f',\phi,D)\in out(u)} ?\phi; D; \mathsf{pc} := n'; ?\mathsf{capacitor} \geq \mathit{Qmin}_i$$

$$\delta\mathsf{jmp} \equiv \cap_{(n',f',\phi,D)\in out(u)} ?\phi^d; D^d; \mathsf{pc} := n'; ?\mathsf{capacitor} \geq \mathit{Qmin}_i{}^d$$

$$\mathsf{getc} \equiv \mathsf{capacitor} := q_{\mathsf{qp}}; \mathit{Qmax}_{\mathit{tmp}} := \mathit{Qmax}_{\mathsf{qp}}$$

$$\mathsf{pickc} \equiv \cup_{j\in[1,\mathsf{nQ}]} \mathsf{qp} := j$$

$$\mathsf{maxc} \equiv \mathsf{capacitor} := \min(\mathsf{capacitor}, \mathit{Qmax}_{\mathit{tmp}})$$

$$\mathsf{setc} \equiv q_{\mathsf{qp}} := \mathsf{capacitor} \qquad \mathsf{QC} \equiv \mathit{Qmin}_i \leq \mathsf{capacitor}$$

$$\alpha\Box f \equiv ?\mathsf{pc} = n; \{f \& P' \wedge \mathsf{QC}\}; \mathsf{maxc}; ?P'$$

$$\delta\Box f \equiv ?\mathsf{pc} = n; \{f \& \mathsf{QC}\}^d; \mathsf{maxc}; ?P'$$

$$\alpha\Diamond f \equiv ?\mathsf{pc} = n; \{f \& \mathsf{QC}\}; \mathsf{maxc}$$

$$\delta\Diamond f \equiv ?\mathsf{pc} = n; \{f \& \mathsf{QC}\}^d; \mathsf{maxc}$$

$$\alpha \mathit{lastf} \equiv \{f \& \mathsf{QC} \wedge \Phi\} \text{ where}$$
$$\Phi \equiv \bigwedge\{\neg\phi \mid ((n,f),v,\phi,D) \in E\} \text{ for current } n, f$$

$$\{u_1,\ldots,u_n\} \rightsquigarrow_G \mathsf{pc} = u_1 \vee \cdots \vee \mathsf{pc} = u_n$$

$$op(P_1,\ldots,P_n) \rightsquigarrow_G op(P_1',\ldots,P_n') \text{ for 1st-order } \textit{op}\text{erator}$$
$$\text{where } P_i \rightsquigarrow_G P_i' \text{ for each } i \in [1,n]$$

$$\Box P \rightsquigarrow_G \Big\langle \Big\{ \mathsf{getc}; \{ \{\mathsf{pickc}\} \cup \{\{\cup_{(n,f)\in\mathsf{VA}}\{\alpha\mathsf{jmp}; ?P' \cup \alpha\Box f\}\} \cup \{\cap_{(n,f)\in\mathsf{VD}}\{\delta\mathsf{jmp}; ?P' \cap \delta\Box f\}\}\} ; \mathsf{setc}\Big\}^{\times}\Big\rangle P' \quad \text{where } P \rightsquigarrow_G P'$$

$$\Diamond P \rightsquigarrow_G \Big\langle \Big\{ ?\neg\mathsf{pc} = \mathsf{done}; \mathsf{getc}; \{ \{\mathsf{pickc}\} \cup \{\{\cup_{(n,f)\in\mathsf{VA}}\{\alpha\mathsf{jmp} \cup \alpha\Diamond f\}\} \cup \{\cap_{(n,f)\in\mathsf{VD}} \{\delta\mathsf{jmp} \cap \{\delta\Diamond f \cup \alpha \mathit{lastf}; ?P'; \mathsf{pc} := \mathsf{done}\}\}\} \cup \{?P'; \mathsf{pc} := \mathsf{done}\}\} ; \mathsf{setc}\Big\}^{*}\Big\rangle(\mathsf{pc} = \mathsf{done} \wedge P')$$

Fig. 9. Translation into CdGL

holds at all times, translates to a game where Demon controls the repetition of the main loop ($^{\times}$). The loop body starts by initializing temporary variables with capacitor information, then proceeds to a choice game with three branches, which respectively implement capacitor-switching, Angelic transitions, and Demonic transitions. The bodies of each branch are built up from auxiliary definitions that implement each rule from the trace semantics (Section V), after which temporary capacitor variables are written back to non-temporary variables (setc). The branches for Angelic and Demonic transitions are defined as choices across nodes, but the auxiliary definitions contain tests on the current task name pc so that only the current task's transitions may be applied. Note that the postcondition of the game is the P': the postcondition must hold at the end of each loop repetition. Tests are inserted into the game branches to test P' at key moments. In doing so, we test an even stronger condition: P' holds at all times throughout execution.

The translation of $\Diamond P$ also follows a loop which begins with getc, albeit an Angelic loop. The body contains branches for every applicable transition rule, yet it features an extra nuance: what happens if P becomes true during a Demonic task? We model this case by allowing Angel to pick the duration of a Demonic task if they commit to ending the loop and proving P' after the current iteration. We write done for some fixed value disequal to every task identifier, and use $\mathsf{pc} = \mathsf{done}$ to indicate that we have committed to terminating the loop after the current execution. The postcondition $\mathsf{pc} = \mathsf{done}$ requires us to run the loop until termination. On the flip side, execution must not continue past when $\mathsf{pc} = \mathsf{done}$ occurs. In short, the loop is exactly a `while` loop with the condition $\mathsf{pc} \neq \mathsf{done}$. We are permitted to terminate the loop at any iteration where the postcondition P' is already true ($?P'; \mathsf{pc} := \mathsf{done}$). Otherwise, we apply transition rules normally. For continuous Demonic transitions, Angel chooses whether to let Demon control the task normally, vs. taking control, proving the postcondition, and ending the loop. If Angel takes control and ends the loop early, their transition must end before any Demonic transition guard becomes true, otherwise Demon would gain the right to transition regardless of Angel's wishes. In contrast to the translation of $\Box P$, Angel is not required to prove the postcondition P' true throughout the evolution of physics, but only after deciding to terminate the loop.

Note that a complete LTL specification for a program specifies both its *theorem* and the *initial values*, thus the implementation of BatFly outputs the translation of $(P \to Q)$ where P comes from the SinitialValues declaration and Q comes from Stheorem.

Why is done required for $\Diamond P$ but not $\Box P$? In $\Diamond P$, there need only *exist* a time that P holds during a Demonic ODE; *Angel* gets to pick which time. We implement Angel's ability to interrupt Demon's final ODE execution with use done; interruption is not built in to CdGL. In contrast, the notion of "all times" already has direct support: domain constraints and tests.

For the sake of technical precision, we note that the CdGL syntax does not explicitly support indexed assignments such as $q_i := e$ or $x := q_i$ where i is an *arbitrary term* rather than a *constant number*. However, we use this syntax because it is *definable* from n-ary choices and tests in CdGL, where nQ is the number of capacitors:

$$q_i{:=}e \equiv \cup_{j\in[1,\mathsf{nQ}]} ?i{=}j; q_j{:=}e \qquad x{:=}q_i \equiv \cup_{j\in[1,\mathsf{nQ}]} ?i{=}j; x{:=}q_j$$

Our translation is non-trivial because temporal correctness properties under adversarial environments are non-trivial. Because such properties are important to the correctness of ICS programs, it is worthwhile to develop such translations. Such translations potentially have theoretical impact beyond ICS programs, as we are not aware of any prior work that explicitly connects LTL-style reasoning on games to game logics such as CdGL. Our translation's complexity shows i) that ICS models deserve formal semantics (Section V), ii) that the translation deserves a correctness proof (Section VIII), iii) and that ICS deserve a custom modeling language, BatFly. BatFly models (Figure 5) are significantly simpler than CdGL ones because the translation handles details of turn-taking, the program counter, capacitor management, early termination, universal temporal properties, exhaustiveness, and assignment indexing automatically. Thus, reduced bugs and labor are predicted. Our prototype implementation of BatFly automatically parses the concrete syntax (Section III-B), converts it to abstract syntax (Section III-C), applies the translation to CdGL, then outputs a model usable by KeYmaera X (Section VII).

VII. Proving ICS in dGL

Now that our program's LTL specification has been translated to hybrid games, we can perform a hybrid game proof to guarantee (by Section VIII) that the original program meets its original LTL specification. This section introduces a generic proof paradigm for ICSs, then applies the paradigm to the CEM example.

a) Proof Paradigm: Amortized Energy Analysis: Intermittence greatly complicates program semantics: a program might restart many times, with significant wasted work. Our key proof insight is that program *verification* need not gain such complexity. Verification seeks to show there *exists* an Angelic control strategy for the ICS that ensures its correctness property for *all* Demonic environments.

Because restarts equate to wasted work, a strategy with restarts is always dominated by a strategy without them. Thus we lose no expressive power, and gain great simplicity, by restricting ourselves to proofs for restart-free strategies.

Correctness is also proved rather than assumed, so we require proof that execution is restart-free. Proving restart-freedom is the fundamental challenge that ICS verification adds to standard CPS verification. We have identified the fundamental proof technique for restart-freedom: amortized energy analysis. An ICS slowly accumulates energy over a long period of time, then spends energy in a short burst. Just as amortized algorithms budget the cost of a few expensive operations across many cheap operations, we budget the energy spent during short bursts against the charge cycle to ensure that energy is never exhausted. We identify the maximum energy spent in a burst, identify the charging rate, and observe that the time between bursts is always sufficient to charge all capacitors to necessary voltage. Budgeting is nontrivial because capacitors discharge linearly but charge exponentially. Under exponential charging, the optimal strategy often requires distributing energy across several capacitors. Once the budget is identified, we prove that each task obeys its budget. The proof for each task uses pre-existing, albeit cutting-edge [9], hybrid systems proof techniques.

b) Technicalities: Until now, we have used CdGL as our logic for hybrid game proofs, but it is not the only one: differential game logic (dGL) is another logic with the exact same formulas and games. The differences between CdGL and dGL are small and technical: to promote a strong connection between proofs and executable code, CdGL prevents one from assuming that numbers can be compared exactly ($x \leq y \vee x > y$). We have used CdGL due to that connection and in particular because it enables the proofs of Section VIII.

However, dGL has a more mature theorem-prover (KeYmaera X [12]) than CdGL (Kaisar [7, Ch. 7]). Thus, we now use dGL and KeYmaera X, but it is safe to switch between the two logics when nonconstructive proof techniques are avoided [7], which we have done, e.g., our proof does not rely on exact numeric comparisons.

c) CEM Example: We prove the CEM model Figure 5 in KeYmaera X after exporting it to KeYmaera X format using the BatFly prototype. We prove $\Box(\neg\{spoiled\} \wedge temp < 8)$, meaning we always avoid the spoiled state and the spoilage temperature. Just as the CEM model illustrated general-purpose modeling constructs, the CEM proof is illustrative of general-purpose proof techniques. Specifically, the proof illustrates the amortized energy analysis technique which is applicable to any ICS. More broadly, the proof is an invariant proof, and invariant proofs are the canonical approach for *any* safety property $\Box P$ in *any* repeated system, ICS or otherwise. An invariant proof for a game proves $\Box P$ by picking an invariant formula J that implies P, picking an Angel strategy, and showing the strategy preserves J before and after each loop repetition, for every Demon. Thus the fundamental proof steps are identifying a strategy and identifying an invariant J.

Identifying a strategy consists of identifying Angel's choices of task transitions, capacitor switches, and ODE duration. We seek a strategy that ensures restart-freedom, preserves safety, and uses only transitions allowed by the model. Thus, energy budgeting is a guiding principle for strategy design, together with safety. We summarize our chosen strategy before presenting the invariant. Demon promises at least 12 hours between alerts. Angel spends this time in the `slow` state to charge safely, charging the capacitors for 1 and 11 hours, respectively. Angel switches to `fast` in order to sense alerts quickly, running on capacitor 2, the only one with sufficient energy for the `fast` state. For simplicity, the model lets Angel trigger the alert, so we switch to `alert` after 1 minute. Angel switches to capacitor 1 because capacitor 2 is near-empty. Demon picks any duration up to 3 minutes, but Angel has enough energy for every case.

We present the invariant of our proof in Figure 10. The invariant is a disjunction with one branch for each stage of the strategy. Each invariant branch must be strong enough to show both restart-freedom and safety. The former is achieved with bounds on capacitor voltage, the latter with bounds on temperature. In the invariant, recall that pc stands for the current task name, qp indicates the active capacitor, and other variables are as defined by Figure 5. Each branch provides bounds on the same variables. The first 7 branches are given as a table. The 8th is Demonic, thus time-dependent, and listed separately for space reasons. We derived the invariant in Figure 10 by

$$J \equiv b_1 \wedge b_2 \wedge b_3 \wedge b_4 \wedge b_5 \wedge b_6 \wedge b_7 \wedge b_8$$

$$b_i \equiv \mathsf{pc} = \mathsf{pc}_i \wedge \mathsf{qp} = \mathsf{qp}_i \wedge \mathit{temp} \in t_i \wedge \mathit{capacitor}_1 \in c_{(1,i)} \wedge \mathit{capacitor}_2 \in c_{(2,i)} \wedge \mathit{time}_i(\mathit{clock}, \mathit{stopwatch}) \quad \text{for } i \in [1,7]$$

$$b_8 \equiv \mathsf{pc} = \mathit{alert} \wedge \mathsf{qp} = 1 \wedge \mathit{temp} \in [-73.366 + \mathit{stopwatch} * 1.53369, -20.050 + \mathit{stopwatch} * 8.67294] \wedge \mathit{capacitor}_1 \in [(43.8/7) - 1.4\ \mathit{stopwatch}, 7] \wedge \mathit{capacitor}_2 \in [1.8, 7] \wedge \mathit{stopwatch} \in [0, 3]$$

i	pc$_i$	qp$_i$	t_i	$c_{(1,i)}$	$c_{(2,i)}$	$time_i(cl, s)$
1	slow	1	[-90,-60]	[1.8,7]	[1.8,7]	$cl{=}s{=}0$
2	slow	1	[-87.612,-56.61]	[43.8/7,7]	[1.8,7]	$cl{=}s{=}60$
3	slow	2	[-87.612,-56.61]	[43.8/7,7]	[1.8,7]	$cl{=}s{=}60$
4	slow	2	[-73.387,-20.108]	[43.8/7,7]	[563.8/7,7]	$cl{=}s{=}720$
5	fast	2	[-73.387,-20.108]	[43.8/7,7]	[563.8/7,7]	$cl{=}720, s{=}0$
6	fast	2	[-73.366,-20.050]	[43.8/7,7]	[1.8,7]	$cl{=}721, s{=}1$
7	alert	2	[-73.366,-20.050]	[43.8/7,7]	[1.8,7]	$s = 0$

Fig. 10. Proof invariant.

augmenting our strategy with temperature and voltage bounds. The formal KeYmaera X proof uses the latest ODE invariant and variant techniques to rigorously prove both upper and lower bounds for both Angelic and Demonic ODEs by bounding derivatives (similar to b_8) and duration and by determining equilibria. In future work, proof automation could be explored which computes restart-free duration and derives bounds for each duration.

The invariant proof step is the most important. Also important are three proof steps specifying the duration of the `slow` and `fast` steps. In particular, Angelic control of ODE duration was essential because it is only by switching capacitors during the `slow` task that Angel can accrue sufficient energy to meet future demand. The full KeYmaera X proof has 623 proof steps input by the user. Of the remaining 619 steps, dozens help prove rigorous variable bounds, but hundreds are boilerplate such as executing code, expanding definitions, and selecting which invariant branch to prove. Though inputting the actions takes tens of minutes, KeYmaera X version 4.9.9 can perform the checking in tens of seconds on a modern workstation. Our experience suggests that although invariants and bounds can be determined systematically, performing a formal proof in KeYmaera X currently requires significant user expertise. The current need for expertise, combined with the high number of boilerplate steps, suggests special-purpose proof automation for BatFly might improve productivity in future work. Because invariants and particularly amortized energy analysis can be used in any ICS safety proof, it is expected that our illustrative example proof could be generalized to create such automation. Though KeYmaera X proofs remaining challenging, that makes it all the more beneficial that we structured our approach to let proof experts do proofs and let ICS experts design ICSs.

VIII. Theory: Soundness

We showed how to verify an ICS program by translating it into a hybrid game and verifying the game. This section shows that when

the game is verified, the ICS program truly does meet its specification. This is because the translation is *sound*: the game does the same thing as the ICS program (simulates it), so properties of the game extend to the ICS program. **Soundness** means that if the CdGL translation is valid then the LTL formula is valid for the ICS program, where validity is defined in Definition II.1 and Definition V.2.

Theorem VIII.1 (Soundness Theorem)**.** *For all graphs G and all LTL formulas P, if $P \rightsquigarrow_G P'$ and $\models_{CdGL} P'$ then $\models_G P$.*

Sketch. We condense our proof to a sketch due space constraints. The main proof is by induction on the structure of P, with the major cases being $\Box P$ and $\Diamond P$. In each case, the CdGL proof can be expressed in the form $P \rightarrow [\alpha]Q$ for some P, α, Q. Specifically, one can extract a hybrid system [7, Thm. 6.2] called $\boldsymbol{\alpha}$ which *refines* α, [7, Thm. 6.4] e.g., by playing one of its winning strategies. Because the hybrid system is in *normal form*, one can traverse it and reconstruct Angel's implementation $AI()$ which is needed by $\models_G P$.

By [7, Thm. 6.3], $\boldsymbol{\alpha}$ satisfies the same postcondition as P. To show that G satisfies the same postcondition under $AI()$, it suffices to show G is mathematical *simulation* of $\boldsymbol{\alpha}$. That is, our proof defines what it means for a BatFly machine m and CdGL state ω (recall: ω is a collection of real-number values $\omega(x)$ for each variable) to be the same, then show that transitions of G and $\boldsymbol{\alpha}$ take same states to same states. The proof of such consists of a lemma for each transition rule showing that it correctly implements the corresponding branch of the hybrid game. Thereupon, we conclude that because G has traces contained in those of $\boldsymbol{\alpha}$, and P holds at each moment of the trace, then $\Box P$ is valid for G with $AI()$.

The $\Diamond P$ case is similar but additionally ensures execution of G ends at the same iteration as $\boldsymbol{\alpha}$. □

Theorem VIII.1 reflects the fundamental insight that CPS proofs and restart-freedom proofs enable ICS proofs. We neither claim nor prove that every execution of an ICS is represented in the hybrid game translation; specifically, executions with restarts are not. The fundamental claim is rather that restart-freedom proofs enable a powerful CPS-driven proof method for ICS.

IX. Related Work

We discuss languages for IC, hybrid games, and timed systems.

Languages and Runtimes for Intermittent Computing. Programming languages for ICS are well-studied, but ICS languages with cyber-physical guarantees are not. Our use of energy-aware cyber-physical models and proofs contrasts with all of the following, and is fundamental. Proofs of physical properties require detailed physical models and physical assumptions, which competing approaches do not provide. Many related works help programs behave reasonably when assumptions and guarantees fail; thus these works are useful safety-nets for practical application of BatFly. DINO [2] and Alpaca [3] have formal semantics, even formalized in a theorem-prover [18], but they respectively focus on using checkpointing and task-based execution to provide robust programming abstractions, the latter introduced in the language Chain [13]. Capybara [19] uses capacitor arrays to improve performance under peak load. Energy types [20] help a programmer track which code is believed to have high or low energy cost, but do not guarantee sufficient energy. We have formal semantics and a task-based approach that supports capacitor arrays, but our proof contributions lie in identifying and demonstrating a generic approach for proving ICSs correct by combining CPS correctness proofs with restart-freedom proofs.

CleanCut [21] ensures termination of task-based programs by subdividing tasks. Compared to BatFly, it supports a richer probabilistic energy model, but only ensures a far narrower correctness property: programs terminate, but with no guarantee on physical behaviors.

MayFly [4] is the namesake of BatFly; both are coordination languages which can be combined with another language for implementation. MayFly recognizes that values are time-sensitive, so it lets the programmer express constraints wherein old values expire or new values must be sensed regularly. However, it has no model of physics, thus precluding the proofs that BatFly provides.

Logics of Hybrid Games and Systems. Hybrid games [6] can be proved correct using two closely-related versions of formal logic: dGL [22] and CdGL [7, Ch. 5], which are respectively implemented in the KeYmaera X [12] theorem-prover and Kaisar [7, Ch. 7] proof language. This paper uses dGL and CdGL near-interchangeably because the technical difference between them is minor: comparison formulas in CdGL use *constructive reals* and specifically use a version of comparison that translates cleanly to executable code. Our theoretical results (Section VIII) use CdGL because it supports advanced refinement-based proof techniques [7, Ch. 6], but our example proof (Section VII) uses dGL because the implementation of KeYmaera X is more mature than Kaisar. Both KeYmaera X [7, Ch. 3] and Kaisar [7, Ch. 8] can extract executable code from models and proofs, but only Kaisar can extract code for hybrid *games*. Both dGL and CdGL are descendants of *differential dynamic logic* [23] (dL), a logic for hybrid systems [24], which are 1-player hybrid games. BatFly follows dGL's tradition of making hybrid games as general as possible. Take two examples: Angelic ODE duration control is supported and arbitrary turn-taking patterns are supported, including multiple Angelic actions per Demon action or even total Angelic control. Rich turn structure is common and natural for task-based ICS because one turn can include multiple tasks. For example: Angel might switch between `slow` and `fast` within the same turn. Such games are strictly more powerful than certain commonly-used fragments, such as strict turn alternation commonly used in synthesis or the common use of hybrid systems (1-player games) in dL theorem-proving. In such cases, added power does no harm; a BatFly user is welcome to use just their desired fragment.

Numerous logics study LTL-like properties on games; Alternating-Time Temporal Logic [17] is one such approach, and provides a good overview of the others. Among the well-established approaches, our version of LTL uses one often called the *supervisory control* approach: the LTL formula is valid when there exists a strategy under which a given LTL strategy is satisfied. Our LTL is notable, however, for its support for general strategies for general hybrid games. For example, a strategy need not be limited to discrete control code, and can include even strategies for continuous parts of the system. Our LTL can express properties about real-valued variables, similar to logics like STL [25], but does not directly support STL's generalized $\Box P$ and $\Diamond P$ formulas which respectively capture properties that remain true for a given amount of time or become true within a given amount of time, since we have not yet needed this feature.

Temporal Logic and Synthesis. LTL is widely used for specifying temporal properties [5] including safety and liveliness. Games have been used to synthesize controllers for certain hybrid systems [26] that meet given LTL safety properties, but in general hybrid systems [23] and therefore games are undecidable, with games synthesis semidecidable only for highly restricted subproblems [27]. Timed games are hybrid games where all variables evolve at the same constant rate and switched systems are hybrid systems where the only discrete dynamics is switching between different ODEs. Synthesis for safety properties of timed game automata is well-studied [28], as is synthesis for controlled reachability of switched system automata [29] but both classes of systems are too restrictive for even the

piecewise-constant ODE from our example model. These automata-based approaches do not support general LTL properties for general hybrid games with polynomial ODEs. Compared to such approaches, BatFly's key strengths are (1) its support for such properties and (2) its higher-level modeling language, at the cost of requiring proofs.

X. Current Limitations and Future Works

We discuss our work's limitations and how to address them.

- Our model is not evaluated on real hardware. Fortunately, the proof tool Kaisar [7, Ch. 7] supports automatically extracting code from proofs and models [7, Ch. 8]. We developed and analyzed a *model* of a temperature monitor. Adding a suitable compiler backend to Kaisar would enable executing on hardware and pursuing an *end-to-end* verification approach: validating that model assumptions hold at runtime, ensuring safety.
- We defer verification of probabilistic ICS models to future work because it is known to require fundamentally different techniques that deserve separate study [30], [31]. We study long-term behavior, where averages are good approximations, especially if harvesters provide steady power [32].

Lastly, one apparent limitation of the approach is not a limitation, but a strength: we prove an ICS correct by analyzing cases where uncontrolled loss of power does not occur. This does not mean we have ignored their fundamental trait, intermittence, but rather that we have analyzed a system's power needs to ensure that its average-case energy availability suffices for restart-freedom.

XI. Conclusion

Intermittently powered embedded systems (ICSs) are fundamental to a variety of embedded application domains including IoT devices, medicine, and infrastructure monitoring. Correctness of ICSs is typically a cyber-physical property: control code must control the system in such a way that its physical evolution over time is correct, despite intermittence. Yet ICSs are not just any CPS: ICSs face unique energy challenges, for which we introduced restart-freedom proofs using our new amortized energy analysis proof technique. These correctness properties arise across a range of applications and are frequently critical to either safety or quality-of-service, thus it is necessary to develop a general-purpose, rigorous verification approach for cyber-physical properties of ICSs.

This paper contributed such an approach, which, to the best of our knowledge, is the first to formally prove that ICSs meet time-sensitive physical correctness properties under suitable assumptions. Our approach was to introduce a language BatFly for task-based modeling and correctness specification of ICSs. We mathematically defined whether a given BatFly model satisfies a given correctness specification written in LTL. This definition enabled the crucial step: we automatically translate BatFly models to hybrid game models with correctness specifications in CdGL. This step is important because proof software for CdGL and its relatives already exist, thus we have enabled the application of such software to ICSs. Soundness proves such software is applied *correctly*: if existing software claims the hybrid game model is correct, then the BatFly model is correct.

While diverse applications of ICS demand a general approach, it is equally important that general approaches are applied to concrete, well-motivated problems from the start, to demonstrate their impact. Thus, we applied BatFly to temperature monitoring for COVID vaccines. This application highlights that ICSs are important to everyday problems of worldwide interest: several years into the pandemic, vaccine spoilage remains a barrier to vaccine access and thus a threat to public health in many parts of the world. Beyond COVID vaccines, cold storage is critical health infrastructure and will remain an important application over time. By applying BatFly to temperature monitoring, we provided the first fully formal (i.e., machine-checked) proof that a temperature monitor detects temperature changes quickly enough to keep vaccines cold enough to prevent spoilage.

References

[1] P. Sparks, "The route to a trillion devices," *White Paper, ARM*, 2017.
[2] B. Lucia and B. Ransford, "A simpler, safer programming and execution model for intermittent systems," in *PLDI*. ACM, 2015.
[3] K. Maeng, A. Colin, and B. Lucia, "Alpaca: intermittent execution without checkpoints," *Proc. ACM Program. Lang.*, no. OOPSLA, 2017.
[4] J. Hester, K. Storer, and J. Sorber, "Timely execution on intermittently powered batteryless sensors," in *SenSys*. ACM, 2017.
[5] A. Pnueli, "The temporal logic of programs," in *FOCS*. IEEE, 1977.
[6] A. Nerode, J. B. Remmel, and A. Yakhnis, "Hybrid system games: Extraction of control automata with small topologies," in *Hybrid Systems*, ser. LNCS. Springer, 1996.
[7] R. Bohrer, "Practical end-to-end verification of cyber-physical systems," Ph.D. dissertation, Carnegie Mellon University, 2021.
[8] A. F. Santos, P. D. Gaspar, and H. J. de Souza, "Refrigeration of COVID-19 vaccines: ideal storage characteristics, energy efficiency and environmental impacts of various vaccine options," *Energies*, 2021.
[9] Y. K. Tan and A. Platzer, "An axiomatic approach to existence and liveness for differential equations," *Formal Aspects Comput.*, 2021.
[10] L. Taylor, "Why Cuba developed its own COVID vaccine—and what happened next," *British Medical Journal*, vol. 374, 2021.
[11] K. P. Acharya, T. R. Ghimire, and S. H. Subramanya, "Access to and equitable distribution of COVID-19 vaccine in low-income countries," *npj Vaccines*, vol. 6, no. 1, 2021.
[12] N. Fulton, S. Mitsch, J.-D. Quesel, M. Völp, and A. Platzer, "KeYmaera X: An axiomatic tactical theorem prover for hybrid systems," in *CADE*, ser. LNCS. Springer, 2015.
[13] A. Colin and B. Lucia, "Chain: tasks and channels for reliable intermittent programs," in *OOPSLA*. ACM, 2016.
[14] B. Islam and N. Shahriar, "Scheduling computational and energy harvesting tasks in deadline-aware intermittent systems," in *RTAS*. IEEE, 2020.
[15] D. R. Biba, M. C. Ancuti, A. Ianovici, C. Sorandaru, and S. Musuroi, "Power supply platform and functional safety concept proposals for a powertrain transmission electronic control unit," *Electronics*, vol. 9, no. 10, p. 1580, 2020.
[16] W. Walter, *Ordinary Differential Equations*, ser. Graduate Texts in Mathematics. Springer, 1998, vol. 182.
[17] R. Alur, T. A. Henzinger, and O. Kupferman, "Alternating-time temporal logic," in *FOCS*. IEEE, 1997.
[18] M. Surbatovich, B. Lucia, and L. Jia, "Towards a formal foundation of intermittent computing," *OOPSLA*, 2020.
[19] A. Colin, E. Ruppel, and B. Lucia, "A reconfigurable energy storage architecture for energy-harvesting devices," in *ASPLOS*. ACM, 2018.
[20] M. Cohen, H. S. Zhu, E. E. Senem, and Y. D. Liu, "Energy types," in *OOPSLA*, 2012.
[21] A. Colin and B. Lucia, "Termination checking and task decomposition for task-based intermittent programs," in *CC*. ACM, 2018.
[22] A. Platzer, "Differential game logic," *ACM Trans. Comput. Log.*, 2015.
[23] ——, "Differential dynamic logic for hybrid systems." *J. Autom. Reas.*, vol. 41, no. 2, 2008.
[24] A. Nerode and W. Kohn, "Models for hybrid systems: Automata, topologies, controllability, observability," in *Hybrid Systems*, ser. LNCS, vol. 736. Springer, 1992.
[25] A. Donzé and O. Maler, "Robust satisfaction of temporal logic over real-valued signals," in *FORMATS*, ser. LNCS, vol. 6246. Springer, 2010. [Online]. Available: https://doi.org/10.1007/978-3-642-15297-9_9
[26] C. J. Tomlin, J. Lygeros, and S. S. Sastry, "A game theoretic approach to controller design for hybrid systems," *Proc. IEEE*, 2000.
[27] O. Shakernia, G. J. Pappas, and S. Sastry, "Semi-decidable synthesis for triangular hybrid systems," in *HSCC*. Springer, 2001.
[28] E. Asarin, O. Maler, A. Pnueli, and J. Sifakis, "Controller synthesis for timed automata," *IFAC Proceedings Volumes*, vol. 31, no. 18, 1998.
[29] A. Taly and A. Tiwari, "Switching logic synthesis for reachability," in *EMSOFT*. ACM, 2010.
[30] J. Y. Halpern, *Reasoning about uncertainty*. MIT Press, 2005.
[31] A. Platzer, "Stochastic differential dynamic logic for stochastic hybrid programs," in *CADE*, ser. LNCS, vol. 6803. Springer, 2011.
[32] G. Gobieski, N. Beckmann, and B. Lucia, "Intelligence beyond the edge: Inference on intermittent embedded systems," *ASPLOS*, 2019.