# Facilitating Concurrency in Hybrid Programs

Haithem Turki*
Carnegie Mellon University
hturki@andrew.cmu.edu

Long Pham*
Carnegie Mellon University
longp@andrew.cmu.edu

December 7, 2019

**Abstract**

Cyber-physical systems (CPSs) are inherently composite—by definition, a CPS comprises at least two distinct components for discrete dynamics and continuous dynamics. In the conventional framework of hybrid programs (HPs) and differential dynamic logic (d$\mathcal{L}$), it is necessary to provide a single, monolithic description of an entire cyber-physical system (CPS) before it is ready for formal analysis. This places on the shoulders of users the burden of synthesizing a single hybrid program describing the CPS under consideration even if the system consists of a number of interacting components.

To address this issue, we propose a calculus for concurrent hybrid programs in this paper. Our calculus incorporates the idea of message passing from the well-established process calculus Communicating Sequential Processes (CSP) into hybrid programs. As far as we know, this is the first calculus for concurrent HPs that can handle both of what we call blocking synchronization and non-blocking synchronization. In addition to the formal syntax of the calculus, we present its trace-based denotational semantics and pave the way for a full sequentialization algorithm by presenting key axioms for sequentialization. Finally, we describe a proof-of-work prototype for sequentialization of concurrent hybrid programs that has been built on top of KeYmaera X.

## 1 Introduction

In our project, we wish to extend the existing hybrid programs in the standard differential dynamic logic (d$\mathcal{L}$) to facilitate the modeling and design of a system that consists of multiple interacting and continuously evolving components.

Within the current framework of d$\mathcal{L}$, hybrid programs are written as a sequential sequence of instructions modeling continuous and discrete dynamics. Consequently, in order to model and verify a cyber-physical system (CPS) consisting of multiple components that interact with one another, users are required

*Equal contribution; course section: 15-824

to first integrate all of those interacting components into a single hybrid program before analyzing the resulting composite hybrid program. It is thus incumbent upon the users to take great care when modeling the continuous evolution of the system in a manner consistent with the vast multitude of concurrent phenomena that may be happening in reality at any given point in time.

Furthermore, concurrency in hybrid programs makes it easier to model a CPS with complicated event-triggered/time-triggered control. At present, in d$\mathcal{L}$ and KeYmaera X, users need to specify a single hybrid program that encompasses both the description of physics (i.e. ordinary differential equations; ODEs) and the description of control (i.e. tests and discrete assignments). As a result, this clutters the description of a hybrid program. In event-triggered control, for example, it is necessary to split the evolution domain of an ODE into two or more overlapping regions, resulting in two similar looking ODEs in one hybrid program. Additionally, the lack of ability to separate control from physics demands users to decide how best to combine the continuous and discrete dynamics so as to faithfully mirror the CPS in the users' minds.

Therefore, motivated by (i) the need to model cyber-physical systems with interacting components and (ii) the desire to separate control from physics in a clean way, we propose incorporating message-passing concurrency into hybrid programs. Specifically, our contributions are

1. Adding channels, which allow hybrid programs to communicate with each other by passing messages, to the syntax;
2. Providing trace semantics to the proposed calculus;
3. Devising a sequentialization algorithm that, given multiple interacting hybrid program, produces a hybrid program that describes the global behaviour of the interaction;
4. Extending KeYmaera X to provide a proof-of-concept implementation supporting our augmented syntax.

## 2 Relevant literature

**Process algebras** Process calculi (also known as process algebras) are a family of theoretical frameworks in which we can rigorously reason about concurrent programs. The first process calculus, called Communicating Sequential Processes (CSP), was created by Hoare [1978]. Before the creation of CSP, the only approach to concurrent programming was shared-memory concurrency, where we use low-level constructs (e.g. semaphores and locks) to synchronize programs. In shared-memory concurrency, programs "communicate by sharing memory." However, due to the use of low-level synchronization constructs, it is difficult to write correct concurrent programs in shared-memory concurrency, let alone reasoning about such concurrent programs. By contrast, in CSP, we use channels to pass information across interacting programs/processes. Hence, in effect, programs "share memory by communication" rather than "communicating

by sharing memory." This paradigm of concurrency is called message-passing concurrency and is more amenable to formal analysis than shared-memory concurrency.

CSP has been influential in the design of many practical programming languages, including Golang by Google, and other process algebras such as Calculus of Communicating Systems (CCS) by Milner [1980] and the $\pi$-calculus by Milner et al. [1992] (which is a significant extension of CCS where not only constants but also channels can be transmitted across channels).

**Incorporating concurrency into hybrid systems** Various attempts have been made to directly introduce concurrency into the hybrid systems domain. Müller et al. [2018] present a component-based approach that decomposes the overall system into multiple component models whose safety can then be proven individually. Under certain compatibility conditions on how components are connected, the safety of the overall composed system as a whole can then be guaranteed. In order to achieve these guarantees, components interact through interfaces phrased as contracts on the input assumptions and output guarantees that abstract the internal behavior of each individual component. Although this bears similarities to our proposed solution, these contracts abstract the hybrid (continuous-time) behavior of one component to discrete-time observations available to other components.

Platzer [2012] further suggests a dynamic logic (quantified differential dynamic logic; Qd$\mathcal{L}$) for verifying distributed hybrid systems (modeled by quantified hybrid programs; QHP) and an associated proof calculus. The object of study in Qd$\mathcal{L}$ is a system of (possibly an arbitrary/variable finite number of) QHPs that act on global variables (i.e. those variables that are visible to all participating hybrid programs) in an unrestricted manner (i.e. no synchronization mechanisms such as semaphores, locks, and message passing). Equipped with universal and existential quantification, Qd$\mathcal{L}$ allows us to express properties about the global behavior of such a system. Therefore, Qd$\mathcal{L}$ and QHPs are suitable to the study of a cyber-physical system in which all participating hybrid programs are fairly independent of one another and exhibit identical/highly similar behavior. By way of example, consider a system of cars equipped with identical engines. Each car can influence a global variable such as the distance between one car and another car. However, their interaction via global variables is non-blocking in the sense that the behavior of one car never causes other cars to momentarily stop moving. On the other hand, our calculus is tailored to a system consisting of a fixed (and usually small) number of hybrid programs that can communicate with each other by not only acting on global variables but also passing messages around. To sum up, Qd$\mathcal{L}$ is targeted at a large, homogeneous system of somewhat independent hybrid programs, while our calculus suits a small, heterogeneous system of hybrid programs that are interacting more closely with each other.

**Incorporating hybrid systems into process calculi** Instead of adding concurrency to the existing frameworks of hybrid systems, we can work in the reversed direction: build a framework for concurrent hybrid systems by starting with conventional process calculi (e.g. CSP, CCS, and the $\pi$-calculus) and introducing continuous dynamics to them.

One example is hybrid CSP (HCSP) [Jifeng, 1994; Chaochen et al., 1996; Liu et al., 2010], which extends CSP with differential equations and interruptions among several other additional constructs. The semantics of HCSP hinges on the notion of duration that is formalized by the calculus of duration [Chaochen et al., 1991] (more precisely, the extended duration calculus). Additionally, to reason about programs written in HCSP, hybrid Hoare logic has been developed by combining (i) Hoare logic, which employs pre- and post-conditions to reason about imperative programs, and (ii) the calculus of duration, which can handle states that may change continuously over non-zero duration.

On the level of syntax, HCSP is analogous to what we propose in this document: both of them use sequential composition and loops to create hybrid programs from simpler ones. However, these two calculi differ in the types of synchronization they support. Message passing in HCSP can only happen at discrete time in the same manner as in traditional CSP. By contrast, the calculus proposed in this document is capable of "continuously" synchronizing two ODEs that share variables, in addition to the conventional discrete-time synchronization.

Along the same line of research, Schneider and Nestmann [2011] propose hybrid CCS (HCCS), which augments CCS with constructs for continuous dynamics. To define the semantics of HCCS, the authors use what they call continuous trajectories, which are analogous to but distinct from the trace-based semantics we use in this document. In their calculus, two ODEs are required to have the same duration in order to be composed in parallel. On the other hand, in the calculus proposed in this document, we permit parallel composition of ODEs even though they have different durations.

Finally, Rounds and Song [2003] integrate hybrid systems into the $\pi$-calculus, naming the resulting calculus $\Phi$-calculus. Like HCSP, the $\Phi$-calculus differs from our calculus in that the $\Phi$-calculus cannot parallel-compose two concurrent ODEs, while ours is capable of doing so.

# 3 Proposed calculus

## 3.1 Syntax

Fix a set $\mathcal{C}$ of channel names, a set $\mathcal{V}_\ell$ of local variable symbols, and a set $\mathcal{V}_g$ of global variable symbols, where $\mathcal{V}_\ell \cap \mathcal{V}_g = \emptyset$. We propose to extend the existing hybrid program grammar with (i) inputs, (ii) outputs, and (iii) a parallel

operator:

| | |
|---|---|
| $\alpha, \beta ::= x := e$ | assignment |
| $\mid ?Q$ | test |
| $\mid x' = f(x) \,\&\, Q$ | ordinary differential equation; ODE |
| $\mid \alpha \cup \beta$ | nondeterministic choice |
| $\mid \alpha; \beta$ | sequential composition |
| $\mid \alpha^*$ | nondeterministic finite repetition |
| $\mid \alpha \,\|\|\, \beta$ | parallel composition |
| $\mid c!e$ | blocking channel write |
| $\mid c?x$ | blocking channel read |

where the meaning of each symbol appearing in this grammar is

- $\alpha$, $\beta$ are hybrid programs;
- $x \in \mathcal{V}_\ell \cup \mathcal{V}_g$ is a variable;
- $e$ is a polynomial term (including variable symbols);
- $Q$ is a formula of first-order logic of real arithmetic;
- $c \in \mathcal{C}$ is a channel name.

Regarding syntactic precedence, $||$ has higher precedence than $\cup$ but lower precedence than sequential composition (i.e. semicolons).

## 3.2 Intuitive semantics

**Local and global variables** In a system of interacting hybrid programs, local variables $v \in \mathcal{V}_\ell$ are local to each program even if the same variable symbols appear in multiple programs. Global variables $v \in \mathcal{V}_g$, on the other hand, are shared by all participating hybrid programs. Hybrid programs have two ways to modify the values of global variables: assignments and ODEs. If two programs modify global variables by ODEs at the same time, the aggregate gradient of the global variables are given by the sum of individual gradients in the ODEs.

This is a reasonable assumption when regarding gradients as forces/vectors that can independently add up. However, this assumption may not be valid in some application domains. For instance, consider the usage of Amazon Web Service (AWS) EC2 during its 12-month free trial. The cloud service is free of charge during a free trial as long as the usage is limited to 750 hours/month. If we use an ODE to model the cost of using AWS EC2 over the free-trial period, we may write $cost' = 0$, provided that the total usage is under 750 hours/month. Now suppose we have two teams of software engineers that (for some reason) share a single free-trial account of AWS EC2. We can no longer assume that $cost' = 0 + 0 = 0$, since the aggregate usage of EC2 by the two teams may

exceed 750 hours/month even if each individual team can be correctly modeled by $cost' = 0$.

Although this simple problem can be fixed relatively simply by adding more details to the model (specifically, by adding conditionals to the model), variants where the aggregate influence on a global variable is not the sum, but rather the maximum or the average of gradients, are far more difficult to solve with existing semantics.

Lastly, observe that the use of global variables can be seen as (shared-memory) synchronization. We call it non-blocking synchronization to contrast it with blocking synchronization that will be introduced below.

**Channels** Suppose we are given two hybrid programs, $\alpha$ and $\beta$, in our calculus for concurrent cyber-physical systems (CPSs). We sometimes want $\alpha$ and $\beta$ to communicate with each other by passing messages, and this is where channels come in. Given a channel $c \in \mathcal{C}$, $c!e$ means the value of $e$ is sent along channel $c$. Dually, $c?x$ means any value sent across channel $c$ by another process will be placed in variable $x \in \mathcal{V}_\ell \cup \mathcal{V}_g$ of this process. Following the convention of Communicating Sequential Processes (CSP), we use synchronous channels (as opposed to asynchronous channels) where any process that writes to/reads from a channel must wait until there is another process that is willing to engage in communication over this channel. Hence, for example, if $\alpha$ contains $c?x$, $\alpha$ will wait for a value until another process passes a message across $c$.

Intuitively, ! can be regarded as an output operator, and ? can be regarded as an input operator. Although this intuition usually works, the behaviour of channels in process calculi (including CSP and our calculus) can deviate from the intuition. For example, if we have $c!1$ in both $\alpha$ and $\beta$, they will synchronize on this event (and then move on to the next step). According to the above intuition, both $\alpha$ and $\beta$ attempt to send (identical) messages to each other without declaring willingness to receive inputs. However, $\alpha$ and $\beta$ will still be successfully synchronized. Note that if $\alpha$ is about to execute $c!1$ while $\beta$ is about to execute $c!2$, they fail to synchronize, ending up in a deadlock. Therefore, channels should be understood as a means to specify which occurrences of variables in different hybrid programs should have the same value, rather than as (bidirectional) bridges connecting an inputting program and an outputting program. Nevertheless, following the intuitive view of ! and ?, we will occasionally refer to them as the output operator and input operator, respectively, because this is what Hoare [1978] does.

The synchronization enabled by ! and ? is called blocking since all participating programs are required to eventually fire $c!e$ or $c?x$ in order for them to synchronize.

**Connection between hybrid games and message passing** It is worth observing that ? plays the same role as Demon's choice in hybrid games (if you are not familiar with hybrid games, you can safely skip this paragraph). From the viewpoint of $\alpha$, a system involving $\alpha$ and $\beta$ is essentially a game between $\alpha$,

which acts as Angel, and the external environment (i.e. $\beta$), which corresponds to Demon. If $\alpha$ contains $c?x$, $\alpha$ offers (uncountably many) choices on the value (from $\mathbb{R}$) of variable $x$ to the external environment. Therefore, ? can be seen as the dual of nondeterministic/Angel's choice.

## 3.3 Illustrative examples

### 3.3.1 Air conditioning

Consider a room with two air conditioners: $\alpha$ and $\beta$. Suppose they are defined as follows:

$$\begin{aligned}\alpha \equiv & ?(t > T_{\text{hot}}); \{t' = -1\} \cup ?(t < T_{\text{cold}}); \{t' = 3\} \\ \beta \equiv & ?(t > T_{\text{hot}}); \{t' = -3\} \cup ?(t < T_{\text{cold}}); \{t' = 1\},\end{aligned}$$

where $T_{\text{hot}}$ and $T_{\text{cold}}$ are constant and $t$ is a global variable that represents the room temperature. Here, $\equiv$ denotes syntactic equivalence (or definitional equivalence).

We expect their parallel composition to yield

$$\begin{aligned}\alpha \,||\, \beta = & ?(t > T_{\text{hot}}); \{t' = -1\} \,||\, ?(t > T_{\text{hot}}); \{t' = -3\} \\ = & \cup ?(t > T_{\text{hot}}); \{t' = -1\} \,||\, ?(t < T_{\text{cold}}); \{t' = 1\} \\ = & \cup ?(t < T_{\text{cold}}); \{t' = 3\} \,||\, ?(t > T_{\text{hot}}); \{t' = -3\} \\ = & \cup ?(t < T_{\text{cold}}); \{t' = 3\} \,||\, ?(t < T_{\text{cold}}); \{t' = 1\}\end{aligned} \tag{3.1}$$

because parallel composition distributes over nondeterministic choice. The meaning of equality will be formalized in Section 3.4. As explained at the end of Section 3.1, recall that $||$ has higher syntactical precedence than $\cup$ but lower precedence than semicolons.

Next, let us consider the first clause of nondeterminisitic choice. This can be sequentialized into

$$\begin{aligned}& ?(t > T_{\text{hot}}); \{t' = -1\} \,||\, ?(t > T_{\text{hot}}); \{t' = -3\} \\ & = ?(t > T_{\text{hot}}); (\{t' = -1\} \,||\, ?(t > T_{\text{hot}}); \{t' = -3\}) \\ & \quad \cup ?(t > T_{\text{hot}}); (?(t > T_{\text{hot}}); \{t' = -1\} \,||\, \{t' = -3\}).\end{aligned} \tag{3.2}$$

The justification for this step of sequentialization is that we nondeterministically choose between executing the test on the left hand side first and executing the test on the right hand side first.

For the first clause of nondeterministic choice (i.e. executing the left test first), we obtain

$$\begin{aligned}& \{t' = -1\} \,||\, ?(t > T_{\text{hot}}); \{t' = -3\} \\ & = \{t' = -1\}; ?(t > T_{\text{hot}}); \{t' = -1\}; (\{t' = -1\} \,||\, \{t' = -3\}) \\ & \quad \cup ?(t > T_{\text{hot}}); (\{t' = -1\} \,||\, \{t' = -3\})\end{aligned} \tag{3.3}$$

Here, the first clause of nondeterministic choice corresponds to the case where we execute $\{t' = -1\}$ (from the left hand side of parallel composition) before $?(t > T_{\text{hot}})$ (from the right hand side of parallel composition). In this situation, we obtain the expression $\{t' = -1\}; ?(t > T_{\text{hot}}); \{t' = -1\}$ because the test $?(t > T_{\text{hot}})$ may happen (i) during the execution of ODE $\{t' = -1\}$ or (ii) after the ODE has finished. A crucial observation to make here is the duration of ODE can be zero. Hence, $\{t' = -1\}; ?(t > T_{\text{hot}}); \{t' = -1\}$ subsumes these two cases.

Finally, the parallel composition of $\{t' = -1\}$ and $\{t' = -3\}$ is given by

$$\begin{aligned} \{t' = -1\} \,||\, \{t' = -3\} = {} & \{t' = -1\}; \{t' = -4\}; \{t' = -3\} \\ & \cup \{t' = -1\}; \{t' = -4\}; \{t' = -1\} \\ & \cup \{t' = -3\}; \{t' = -4\}; \{t' = -1\} \\ & \cup \{t' = -3\}; \{t' = -4\}; \{t' = -3\}. \end{aligned} \tag{3.4}$$

Here, in the first two clauses of nondeterministic choice, the left ODE starts running before the right ODE, while in the last two clauses, the right ODE runs before the left one. Of the first two clauses, the first one corresponds to the case where the duration of $\{t' = -1\}$ completely subsumes $\{t' = -3\}$, and the second clause is for the case where the left ODE finishes before the right one does (i.e. the two ODEs' durations overlap but neither of them subsumes the other). Note that when two ODEs run simultaneously, the aggregate gradient is given by the sum of the gradients from individual hybrid programs.

To summarize, this example illustrates the following cases for sequentialization of parallel composition:

- Parallel composition over nondeterministic choices (See (3.1));
- Parallel composition of two discrete-time, non-message-passing actions (i.e. tests and assignments) (See (3.2));
- Parallel composition of an ODE and a test (See (3.3));
- Parallel composition of two ODEs (See (3.4)).

### 3.3.2 Event-triggered control

Consider the following two hybrid programs:

$$\begin{aligned} \Phi &= \{x' = v, v' = -g \,\&\, x \geq 0\} \\ \Delta &= ?(x = 0); v := -v \cup ?(x \neq 0). \end{aligned}$$

$\Phi$ describes the physics of a ball's free fall ($\Phi$ is chosen because its pronunciation is similar to"physics"). $\delta$ (no particular reasoning behind the choice of this Greek letter) describes an event-triggered control system that bounces back the ball upon the ball's hitting the ground and otherwise does nothing.

Now we would like to have these two programs interact with each other by both non-blocking synchronization (i.e. global variables) and blocking synchronization (i.e. message passing). For simplicity, for the moment, we assume that Control can only bounce back a ball once—this will be amended later. Firstly, because we want both $\Phi$ and $\Delta$ to have access to the position and velocity of the ball, we must have $x, v \in \mathcal{V}_g$. All the other variable symbols can be either local or global.

The next step is to work out where we should put the ? and ! operators. In order to trigger $\Delta$ when the ball hits the ground, we modify the hybrid programs as follows:

$$\begin{aligned}\Phi &= (\{x' = v, v' = -g \,\&\, x \geq 0\}; c!x)^* \\ \Delta &= c?x; (?(x = 0); v := -v \cup ?(x \neq 0)),\end{aligned}$$

where $c \in \mathcal{C}$ is a channel name. As of now, $\Delta$ can have a lapse of time between $?(x = 0)$ and $v := -v$, and in the meantime the ODE in $\Phi$ may keep evolving. To prevent this, we make $\Phi$ wait until it receives a signal from $\Delta$ indicating that $v := -v$ has been executed:

$$\begin{aligned}\Phi &= (\{x' = v, v' = -g \,\&\, x \geq 0\}; c!x; d?y)^* \\ \Delta &= c?x; (?(x = 0); v := -v; \cup ?(x \neq 0)); d!\mathit{done},\end{aligned}$$

where $d \in \mathcal{C}$ is another channel. $\Delta$ sends a message *done* (which can be treated as a value of type String as in programming) simply for the purpose of waking up $\Phi$.

Finally, we make $\Delta$ bounce back the ball whenever it hits the ground:

$$\begin{aligned}\Phi &= (\{x' = v, v' = -g \,\&\, x \geq 0\}; c!x; d?y)^* \\ \Delta &= (c?x; (?(x = 0); v := -v; \cup ?(x \neq 0)); d!\mathit{done})^*.\end{aligned}$$

Their parallel composition, $\Phi \,||\, \Delta$, can be sequentialized into

$$(\{x' = v, v' = -g \,\&\, x \geq 0\}; c!x; (?(x = 0); v := -v \cup ?(x \neq 0)); d!\mathit{done})^*.$$

Now, if we would like the result of sequentialization to look more natural, we can hide all the communication across channels:

$$(\{x' = v, v' = -g \,\&\, x \geq 0\}; (?(x = 0); v := -v \cup ?(x \neq 0)))^*. \tag{3.5}$$

This is indeed what we would write if we were allowed to only use a single hybrid program right from the beginning.

Note that in any successful run of $\Phi \,||\, \Delta$, $\Phi$ and $\Delta$ must repeat the same number of iterations. Otherwise, if one of them stopped before the other, the survived one would indefinitely wait for a partner, falling into a deadlock and hence resulting in an unsuccessful run.

## 3.4 Trace semantics

This section presents the denotational semantics (i.e. each program is associated with a mathematical object) of the calculus we propose. We use trace semantics, where the meaning of an expression is given by a set of traces that represent the history of computation. The formalization and presentation style of trace semantics are inspired by [Platzer, 2010] and [Jeannin and Platzer, 2014].

We start with preliminary definitions.

**Definition 3.1** (State)**.** *Given a set $\mathcal{V}_\ell$ of local variables and a set $\mathcal{V}_g$ of global variables, let $\mathcal{V}$ denote $\mathcal{V}_\ell \cup \mathcal{V}_g$. A state is a mapping $f : \mathcal{V} \to \mathbb{R}$. We denote the set of all all possible states by State($\mathcal{V}$).*

**Definition 3.2** (Action)**.** *An action is one of the following:*

1. *Assignment: $x := e$;*
2. *Test: $?Q$;*
3. *Ordinary differential equation: $(\{x' = f(x) \,\&\, Q\}, r)$, where $r \in \mathbb{R}$ denotes the duration of continuous evolution;*
4. *Input: $c?x$, where $c \in \mathcal{C}$ is a channel name and $x \in \mathcal{V} = \mathcal{V}_\ell \cup \mathcal{V}_g$ is a variable symbol.*
5. *Output: $c!e$, where $c \in \mathcal{C}$ is a channel name and $e$ is a polynomial term.*

**Definition 3.3** (Application of an action to a state)**.** *Given a state $v \in \mathcal{V}$ and an action $a$, $a$ can be successfully applied to $v$ if and only if*

1. *If $a = ?Q$ for some $Q$, $v \models Q$ holds;*
2. *If $a = (\{x' = f(x) \,\&\, Q\}, r)$, we can run the ODE $\{x' = f(x) \,\&\, Q\}$ in the initial state $v$ for duration $r$ without ever exiting the evolution domain $Q$.*

*The state resulting from applying $a$ to $v$ can be defined in a straightforward manner and hence is omitted. Keep in mind that $c?x$ receives a message and sets variable $x$ to the value of the message.*

**Definition 3.4** (Trace)**.** *Fix $\mathcal{V} = \mathcal{V}_\ell \cup \mathcal{V}_g$. A trace is a pair $(v, \sigma)$ of a state $v \in$ State($\mathcal{V}$) and a non-empty, finite sequence $\sigma = (\sigma_1, \ldots, \sigma_n)$ of actions for some $n \in \mathbb{N}$. We say that a trace $\sigma$ terminates (normally) if and only if we can apply the actions $\sigma_1, \ldots, \sigma_n$ sequentially to the initial state $v$ without causing an error.*

**Definition 3.5** (Interleaving of sequences of actions)**.** *Suppose we are given a state $v \in$ State($\mathcal{V}$) and two sequences $\sigma = (\sigma_1, \ldots, \sigma_n)$ and $\rho = (\rho_1, \ldots, \rho_m)$ of actions. The interleaving of $\sigma$ and $\rho$ with respect to $v$, denoted by $v \vdash (\sigma \,||\, \rho)$, is a set of sequences of actions inductively defined below. For brevity, whenever there is symmetry, we only present one of two dual cases.*

1. *If $\sigma$ is empty, $v \vdash \sigma \,||\, \rho$ is a singleton set containing $\rho$.*

2. *Suppose $n, m \geq 1$. If $\sigma_1 = c_1!e_1 \vee \sigma_1 = c_2?x_1$ and $\rho_1 \neq c_2!e_2 \wedge \rho_1 \neq c_2?x_2$, then $v \vdash \sigma \,||\, \rho$ is a singleton set containing $\rho_1 \circ (v' \vdash \sigma \,||\, (\rho_2, \ldots, \rho_m))$, where $\circ$ denotes concatenation and $v'$ is the result of applying $\rho_1$ to $v$, provided that this does not raise an error.*

3. *If $\sigma_1 = c_1!e_1$ and $\rho_1 = c_2!e_2$, then*

$$v \vdash \sigma \,||\, \rho = \{c_1!u \circ (v \vdash (\sigma_2, \ldots, \sigma_n) \,||\, (\rho_2, \ldots, \rho_n))\},$$

*provided that $c_1 \equiv c_2$ (i.e. they are syntactically identical channel names) and $u = v[\![e_1]\!] = v[\![e_2]\!]$ (i.e. $e_1$ and $e_2$ evaluate to the same value under state $v$).*

4. *If $\sigma_1 = c_1!e$ and $\rho_1 = c_2?x$, then*

$$v \vdash \sigma \,||\, \rho = c_1!u \circ (v' \vdash (\sigma_2, \ldots, \sigma_n) \,||\, (\rho_2, \ldots, \rho_n)),$$

*where $c_1 \equiv c_2$, $u = v[\![e]\!]$, and $v'$ is the result of applying $[x := u]$ to state $v$.*

5. *If $\sigma_1 = c_1?x$ and $\rho_1 = c_2?y$, then*

$$v \vdash \sigma \,||\, \rho = \{c_1!u \circ (v' \vdash (\sigma_2, \ldots, \sigma_n) \,||\, (\rho_2, \ldots, \rho_n)) : u \in \mathbb{R}\},$$

*where $c_1 \equiv c_2$ and $v'$ is the result of applying $[x := u]$ and $[y := u]$ (the order of application does not matter) to state $v$. Note that nondeterminism arises in this case. In practice, this case should be extremely rare.*

6. *Assume that each of $\sigma_1$ and $\rho_1$ is either an assignment or a test. We have*

$$\begin{aligned} v \vdash \sigma \,||\, \rho = \{&\sigma_1 \circ (v' \vdash (\sigma_2, \ldots, \sigma_n) \,||\, \rho), \\ &\rho_1 \circ (v' \vdash \sigma \,||\, (\rho_2, \ldots, \rho_m))\}. \end{aligned}$$

7. *Assume $\sigma_1 = (P, r)$, where $P$ is an ODE, and that $\rho_1$ is either an assignment or a test. $v \vdash \sigma \,||\, \rho$ is then a set with two elements:*

$$(P, r_1) \circ \rho_1 \circ (v' \vdash ((P, r_2), \sigma_2, \ldots, \sigma_n) \,||\, (\rho_2, \ldots, \rho_n)), \tag{3.6}$$

*where $r_1, r_2 \geq 0$, $r = r_1 + r_2$, and $v'$ is the result of applying $(P, r_1) \circ \rho_1 \circ (P, r_2)$ to $v$, and*

$$(P, r) \circ (v' \vdash (\sigma_2, \ldots, \sigma_n) \,||\, \rho), \tag{3.7}$$

*where $v'$ is the result of applying $(P, r)$ to $v$. (3.6) corresponds to the case where the discrete-time action $\rho_1$ occurs while the ODE $\sigma_1$ is running, and (3.7) corresponds to the case where $\rho_1$ occurs after $\sigma_1$ finishes.*

8. *Assume $\sigma_1 = (P_1, r_1)$ and $\rho_1 = (P_2, r_2)$. We obtain*

$$\begin{aligned} &v \vdash \sigma \,||\, \rho \\ &= \{(P_1, s_1) \circ (P_1 + P_2, r_2) \circ (v' \vdash ((P_1, r_1 - s_1 - r_2), \sigma_2, \ldots, \sigma_n) \,||\, (\rho_2, \ldots, \rho_m)), \\ &\quad (P_2, s_1) \circ (P_1 + P_2, r_1) \circ (v' \vdash (\sigma_2, \ldots, \sigma_n) \,||\, ((P_2, r_2 - s_1 - r_1), \rho_2, \ldots, \rho_m)), \\ &\quad (P_1, s_1) \circ (P_1 + P_2, r_1 - s_1) \circ (v' \vdash (\sigma_2, \ldots, \sigma_n) \,||\, ((P_2, r_2 - (r_1 - s_1)), \rho_2, \ldots, \rho_m)), \\ &\quad (P_2, s_1) \circ (P_1 + P_2, r_2 - s_1) \circ (v' \vdash ((P_1, r_1 - (r_2 - s_1)), \sigma_2, \ldots, \sigma_n) \,||\, (\rho_2, \ldots, \rho_m))\}, \end{aligned}$$

*where the first element is included if and only if $r_1 > r_2$ and likewise the second element is included if and only if $r_2 > r_1$. Here, the expression $P_1 + P_2$ denotes the ODE obtained by adding gradients on global variables (but not local variables). As usual, $v'$ denotes the result of successful application of the concurrent ODEs to $v$.*

We now present the formal semantics of our calculus for concurrent hybrid programs.

**Definition 3.6** (Trace semantics of conventional hybrid programs)**.** *Let $\alpha$ be a hybrid program without any construct for concurrency (i.e. inputs, outputs, and parallel composition). The semantic object assigned to $\alpha$ is a (possibly uncountably infinite) set $\tau(\alpha)$ of traces defined inductively as follows:*

1. *Assignment: $\tau(x := e) = \{(v, (x := e)) : v \in State(\mathcal{V})\}$.*
2. *Test: $\tau(?Q) = \{(v, (?Q)) : v \in State(\mathcal{V}), v \models Q\}$.*
3. *Ordinary differential equation:*

$$\begin{aligned}\tau(\{x' = f(x) \,\&\, Q\}) = &\{(v, (a)) : v \in State(\mathcal{V}), a = (\{x' = f(x) \,\&\, Q\}, r),\\ &r \in \mathbb{R}, a \text{ can be successfully applied to } v\}.\end{aligned}$$

4. *Nondeterministic choice: $\tau(\alpha \cup \beta) = \tau(\alpha) \cup \tau(\beta)$.*
5. *Sequential composition: $\tau(\alpha; \beta) = \{(v_1, \sigma \circ \rho) : (v_1, \sigma) \in \tau(\alpha), (v_2, \rho) \in \tau(\beta)\}$, where $v_2$ is the result of successful application of $\sigma$ to $v_1$, and $\sigma \circ \rho$ can also be successfully applied to $v_1$. Here, $\sigma \circ \rho$ denotes concatenation of $\sigma$ and $\rho$.*
6. *Repetition: $\tau(\alpha^*) = \bigcup_{n \in \mathbb{N}} \tau(\alpha^n)$.*

**Definition 3.7** (Trace semantics of concurrent hybrid programs)**.** *For the newly introduced constructs, the trace semantics is inductively defined as follows:*

1. *Input: $\tau(c?x) = \{(v, (c?x)) : v \in State(\mathcal{V})\}$.*
2. *Output: $\tau(c!e) = \{(v, (c!e)) : v \in State(\mathcal{V})\}$.*
3. *Parallel composition: $\tau(\alpha \,||\, \beta) = \{(v, \gamma) : (v, \sigma) \in \tau(\alpha), (v, \rho) \in \tau(\beta), \gamma \in v \vdash \sigma \,||\, \rho\}$, where $v \vdash \sigma \,||\, \rho$ is defined in Definition 3.5.*

# 4 Sequentialization algorithm

Converting concurrent programs into their sequential equivalents makes it possible to prove their safety properties using the full suite of existing axioms and tactics already present in KeYmaera X. The sequentialization of concurrent programs in Communicating Sequential Processes (CSP) is a largely understood

phenomenon, with various algebraic laws that allow for the conversion of any concurrent CSP program into its serial equivalent [Hoare, 1985].

In this section, we present several axioms for sequentialization, although the list of our axioms is not exhaustive, since there are a large number of cases to cover (as exemplified by the complexity of Definition 3.5). Also, since these axioms intuitively match the definition of the interleaving of traces in Definition 3.5, the sequentialization axioms in this section seem semantically sound. However, we have not proved the soundness of these axioms; hence, they remain "candidate" axioms rather than legitimate axioms.

The first axiom is the axiom of distributivity over $\cup$:

$$(\alpha_1 \cup \alpha_2) \,||\, (\beta_1 \cup \beta_2) = \alpha_1 \,||\, \beta_1 \cup \alpha_1 \,||\, \beta_2 \cup \alpha_2 \,||\, \beta_1 \cup \alpha_2 \,||\, \beta_2.$$

This axiom is used in the illustrative example in (3.1).

The second axiom is the axiom of parallel composition for two discrete-time, non-message-passing actions (i.e. tests and actions):

$$(a; P) \,||\, (b; Q) = a; (P \,||\, b; Q) \cup b; (a; P \,||\, Q).$$

Regarding !- and ?-actions, one axiom we need is

$$(c?x; P) \,||\, (c!e; Q) = c!e; x := e; (P \,||\, Q).$$

A more formal presentation of the sequentialization axioms for !- and ?-actions can be found in [Hoare, 1985].

Lastly, when two processes start with ODEs, the sequentialization of their parallel composition is given by this axiom:

$$\begin{aligned}
&(\{x' = f(x)\}; P) \,||\, (\{x' = g(x)\}; Q) \\
&= \{x' = f(x)\}; (\{x' = f(x) + g(x)\}; (\{x' = f(x)\}; P \,||\, Q) \\
&\quad \cup \{x' = f(x)\}; (\{x' = f(x) + g(x)\}; (P \,||\, \{x' = g(x)\}; Q) \\
&\quad \cup \{x' = g(x)\}; (\{x' = f(x) + g(x)\}; (P \,||\, \{x' = g(x)\}; Q) \\
&\quad \cup \{x' = g(x)\}; (\{x' = f(x) + g(x)\}; (\{x' = f(x)\}; P \,||\, Q).
\end{aligned}$$

For all the remaining sequentialization axioms, they can be straightforwardly (at least informally) derived from Definition 3.5.

# 5 Implementation

We created a proof of concept implementation of our work by extending KeYmaera X accordingly. We extended the KeYmaera X parser and lexer to handle the necessary extended syntax and enable the authoring of parallel programs. This required introducing not only a new *Parallel* program type, but also propagating the existence of a wholly new *Channel* kind and adding support for it through the entire KeYmaera X code base.

We also implemented a subset of the core and derived axioms needed to prove the safety of parallel programs. Although implementing the full set of axioms is outside the scope of this project, our implementation allowed us to complete proofs of several small examples.

Figure 1: Authoring of parallel programs in KeYmaera X

airConditioner

```
Definitions
  Real Thot;
  Real Tcold;
End.

ProgramVariables
  Real t;
End.

Problem
  (Thot > Tcold)
  ->
  <{{{?(t > Thot); {t'=-1}} ++ {?(t < Tcold); {t'=3}}} || {{?(t > Thot); {t'=-3}} ++ {?(t < Tcold); {t'=1}}} & t}>
  (t <= Thot & t >= Tcold)
End.
```

Figure 2: Parallel axioms suggested in KeYmaera X UI

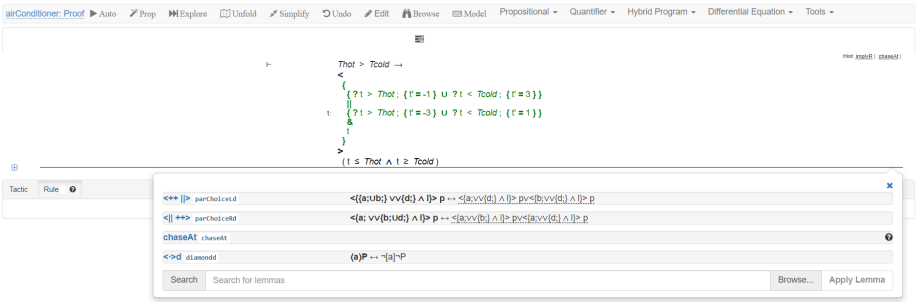
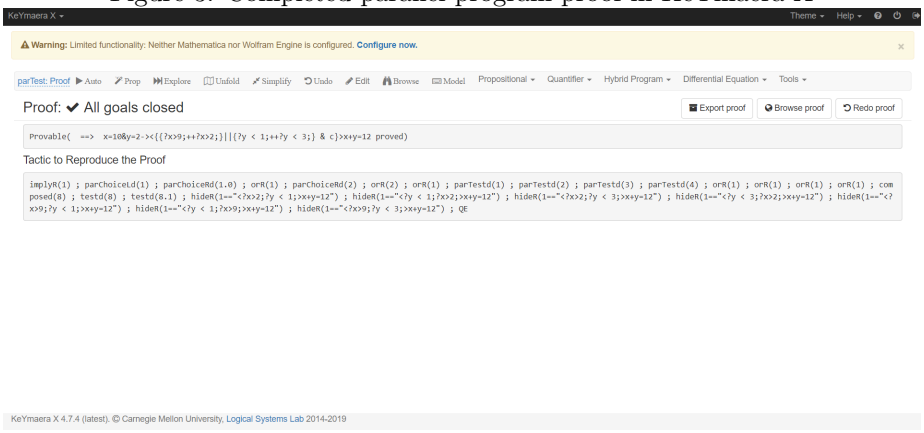


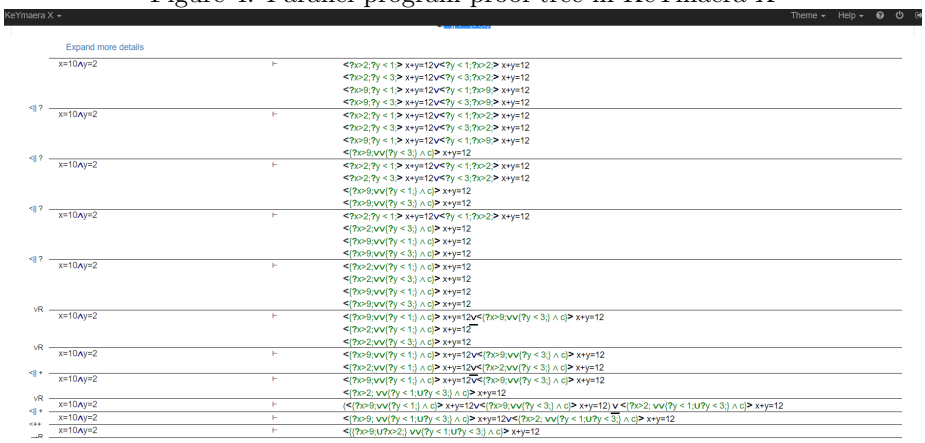

Figure 3: Completed parallel program proof in KeYmaera X




# 6 Conclusion and future work

**Conclusion** Cyber-physical systems (CPSs) are inherently composite—by definition, a CPS comprises at least two distinct components for discrete dynamics

Figure 4: Parallel program proof tree in KeYmaera X




and continuous dynamics. In the conventional framework of hybrid programs (HPs) and differential dynamic logic (d$\mathcal{L}$), it is necessary to provide a single, monolithic description of an entire cyber-physical system (CPS) before it is ready for formal analysis. This places on the shoulders of users the burden of synthesizing a single hybrid program describing the CPS under consideration.

To address this issue, we have proposed a calculus for concurrent hybrid programs in this paper. We have built our calculus by incorporating the idea of message passing from the well-established process calculus Communicating Sequential Processes (CSP) into hybrid programs. As far as we know, this is the first calculus for concurrent HPs that can handle both of what we call blocking synchronization and non-blocking synchronization. In addition to the formal syntax of the calculus, we have provided its trace-based denotational semantics and a couple of notable axioms for sequentialization. Finally, we have built a proof-of-work prototype for sequentialization of concurrent hybrid programs on top of KeYmaera X.

**Future work** With regard to future work, some potential avenues for further research are:

1. Providing a full implementation of our extension in KeYmaera X. Although many of the posible axioms are simple, two challenges are noteworthy. Firstly, providing a robust implementation that can properly sequentialize parallel differential equations will require the creating of arithmetic-aware tactics (to perform arithmetic on common gradients) that differ from the paradigms currently used to implement core and derived axioms. Secondly, we still need to provide formal semantics on how to define free, bound, and must-bound variables in order to properly integrate with the replacement and uniform substitution-based logic underpinning the KeYmaera X codebase.

2. Providing a formal definition of equality based on the trace semantics and formally proving the sequentialization axioms. Although the trace semantics gives an obvious definition of equality (namely equality on traces), we do not think this naïve notion of equality is the right notion for proving the soundness of our sequentialization axioms. We speculate that we probably need a slightly coarser notion of equality. For instance, we would like to identify $?Q$ and $?Q;?Q$ although they technically have different traces.

3. Making our calculus for concurrent hybrid programs more expressive by introducing more constructs. One candidate we specifically have in our mind is a mechanism that wakes up a stalled program when an ODE of another program reaches a certain state. For instance, consider $\alpha$ and $\beta$ defined as

$$\alpha \equiv \{x' = v, v' = -g\}$$
$$\beta \equiv ?(x = 0); v := -v.$$

   As of now, $\beta$ fails (in the sense that it produces an unsuccessful run) whenever the initial state does not satisfy $x = 0$. However, if we can wake up and execute $\beta$ only when the ODE in $\alpha$ reaches $x = 0$, the users no longer need to clutter the code by adding more conditionals. Moreover, the biggest advantage is it is now unnecessary to split evolution domains in event-triggered control. Obviating the need to split evolution domains in event-triggered control was our original motivation for designing a calculus of concurrent hybrid programs, but we found it difficult to achieve this original aim within the limited time frame of this project.

# References


Zhou Chaochen, C.A.R. Hoare, and Anders P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269 – 276, 1991. ISSN 0020-0190. doi: https://doi.org/10.1016/0020-0190(91)90122-X. URL http://www.sciencedirect.com/science/article/pii/002001909190122X. 4

Zhou Chaochen, Wang Ji, and Anders P. Ravn. A formal description of hybrid systems. In Rajeev Alur, Thomas A. Henzinger, and Eduardo D. Sontag, editors, *Hybrid Systems III*, pages 511–530, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg. ISBN 978-3-540-68334-6. 4

C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8): 666–677, August 1978. ISSN 0001-0782. doi: 10.1145/359576.359585. URL http://doi.acm.org/10.1145/359576.359585. 2, 6

C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985. ISBN 0-13-153271-5. 13

Jean-Baptiste Jeannin and André Platzer. dtl2: Differential temporal dynamic logic with nested temporalities for hybrid systems. In Stéphane Demri, Deepak Kapur, and Christoph Weidenbach, editors, *Automated Reasoning*, pages 292–306, Cham, 2014. Springer International Publishing. ISBN 978-3-319-08587-6. 10

He Jifeng. A classical mind. chapter From CSP to Hybrid Systems, pages 171–189. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1994. ISBN 0-13-294844-3. URL http://dl.acm.org/citation.cfm?id=197600.197614. 4

Jiang Liu, Jidong Lv, Zhao Quan, Naijun Zhan, Hengjun Zhao, Chaochen Zhou, and Liang Zou. A calculus for hybrid csp. In Kazunori Ueda, editor, *Programming Languages and Systems*, pages 1–15, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-17164-2. 4

Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980. ISBN 3-540-10235-3. doi: 10.1007/3-540-10235-3. URL https://doi.org/10.1007/3-540-10235-3. 3

Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, i. *Information and Computation*, 100(1):1 – 40, 1992. ISSN 0890-5401. doi: https://doi.org/10.1016/0890-5401(92)90008-4. URL http://www.sciencedirect.com/science/article/pii/0890540192900084. 3

Andreas Müller, Stefan Mitsch, Werner Retschitzegger, Wieland Schwinger, and André Platzer. Tactical contract composition for hybrid system component verification. *International Journal on Software Tools for Technology Transfer*, 20(6):615–643, Nov 2018. ISSN 1433-2787. doi: 10.1007/s10009-018-0502-9. URL https://doi.org/10.1007/s10009-018-0502-9. 3

André Platzer. *Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics*. Springer, Heidelberg, 2010. ISBN 978-3-642-14508-7. doi: 10.1007/978-3-642-14509-4. URL http://www.springer.com/978-3-642-14508-7. 10

Andre Platzer. A Complete Axiomatization of Quantified Differential Dynamic Logic for Distributed Hybrid Systems. *Logical Methods in Computer Science*, Volume 8, Issue 4, November 2012. doi: 10.2168/LMCS-8(4:17)2012. URL https://lmcs.episciences.org/720. 3

William C. Rounds and Hosung Song. The $\phi$-calculus: A language for distributed control of reconfigurable embedded systems. In *Proceedings of the 6th International Conference on Hybrid Systems: Computation and Control*, HSCC'03, pages 435–449, Berlin, Heidelberg, 2003. Springer-Verlag. ISBN 978-3-540-00913-9. URL http://dl.acm.org/citation.cfm?id=1768100.1768134. 4

Sven Schneider and Uwe Nestmann. Rigorous discretization of hybrid systems using process calculi. In Uli Fahrenberg and Stavros Tripakis, editors, *Formal Modeling and Analysis of Timed Systems*, pages 301–316, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-24310-3. 4