ROSCoq: Robots Powered by Constructive Reals

Abhishek Anand(✉) and Ross Knepper

Cornell University, Ithaca, NY 14850, USA
abhishek.anand.iitg@gmail.com

Abstract. We present ROSCoq, a framework for developing certified Coq programs for robots. ROSCoq subsystems communicate using messages, as they do in the Robot Operating System (ROS). We extend the logic of events to enable holistic reasoning about the cyber-physical behavior of robotic systems. The behavior of the physical world (e.g. Newton's laws) and associated devices (e.g. sensors, actuators) are specified axiomatically. For reasoning about physics we use and extend CoRN's theory of constructive real analysis. Instead of floating points, our Coq programs use CoRN's exact, yet fast computations on reals, thus enabling accurate reasoning about such computations.

As an application, we specify the behavior of an iRobot Create. Our specification captures many real world imperfections. We write a Coq program which receives requests to navigate to specific positions and computes appropriate commands for the robot. We prove correctness properties about this system. Using the ROSCoq shim, we ran the program on the robot and provide even experimental evidence of correctness.

1 Introduction

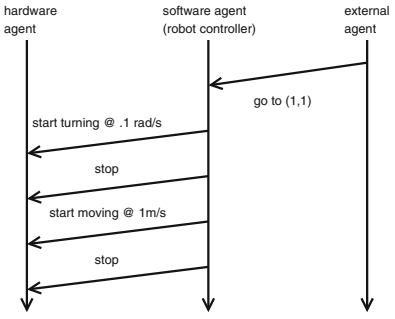
Cyber-Physical Systems (CPS) such as ensembles of robots can be thought of as distributed systems where agents might have sensing and/or actuation capabilities. In fact the Robot Operating System (ROS) [15] presents a unified interface to robots where subcomponents of even a single robot are represented as nodes (e.g. sensor, actuator, controller software) that communicate with other nodes using asynchronous message passing. The Logic of Events (LoE) [3] framework has already been successfully used to develop certified functional programs which implement important distributed systems like fault-tolerant replicated databases [19]. Events capture interactions between components and observations rather than internal state. This enables specification and reasoning at higher-levels while integrating easily with more detailed information [22]. CPSs are arguably harder to get right, because of the additional complexity of reasoning about physics and how it interacts with the cyber components. In this work, we show that an event-based semantics is appropriate for reasoning about CPSs too. We extend the LoE framework to enable development of certified Coq programs for CPSs.

There are several challenges in extending LoE to provide a semantic foundation for CPSs, and thus enable holistic reasoning about such systems: (1) One

C. Urban and X. Zhang (Eds.): ITP 2015, LNCS 9236, pp. 34–50, 2015.
DOI: 10.1007/978-3-319-22102-1_3

has to model the physical quantities, e.g. the position, direction and velocity of each robot and also the physical laws relating them. (2) Time is often a key component of safety proofs of a CPS. For example, the software controller of a robot needs to send correct messages (commands) to the motors before it collides with something. (3) The software controller of a robot interacts with devices such as sensors and actuators which measure or influence the physical quantities. The specification of these devices typically involve both cyber and physical aspects. (4) Robotic programs often need to compute with real numbers, which are challenging to reason about accurately.

Our Coq framework addresses each of these challenges. Our running example is that of a robotic system consisting of an iRobot Create[1] and its controller-software. This setup can be represented as a distributed system with 3 agents (a.k.a. nodes in ROS) : (a) the hardware agent which represents the robot along with its ROS drivers/firmware. It receives messages containing angular and linear velocities and adjusts the motors accordingly to achieve those velocities. (b) the software agent which sends appropriate velocity messages to the hardware agent (c) the external agent which sends messages to the software agent telling where the robot should go. The message sequence diagram below shows a sample interaction between the agents. Click here for the corresponding video.



To define a CPS in ROSCoq, one has to first define its physical model and then define each agent independently. The physical model specifies how the relevant physical quantities evolve over time. These are represented as continuous real-valued functions over time, where time is represented as a non-negative real number. In our example, the relevant physical quantities are the position, orientation and velocities (angular and linear) of the robot. Thanks to dependent types of Coq, it is easy to express physical constraints such as the fact that the velocity is the derivative of the position w.r.t. time. We extensively used CoRN's [8] rich library of definitions and theorems about derivatives, integrals, continuity, trigonometry etc. to represent and reason about physical components. The assumption of continuity allows us to get around many decidability issues related to constructive reals. During the course of this project, we contributed some generic lemmas about constructive real analysis to CoRN, such as a stronger constructive version of Intermediate Value Theorem which we found more useful while reasoning about CPSs[2].

Events have time-stamps and one can specify assumptions on the time needed by activities like message delivery, sensing, actuation, computation etc. to happen. These will have to be empirically validated; currently one cannot statically reason about the running time of Coq programs.

[1] http://www.irobot.com/About-iRobot/STEM/Create-2.aspx.
[2] https://github.com/c-corn/corn/pull/13.

Agents of a CPS are represented as a relation between the physical model (how physical quantities evolve over time) and the trace of observable events (sending and receiving of messages) generated by the agent. This representation allows incomplete and non-deterministic specifications. For hardware devices such as sensors and actuators, this relation is specified axiomatically. For example, the relation for the hardware agent mentioned above asserts that whenever it receives a message requesting a velocity v, within some time δ the robot attains some velocity close to v. The semantics of software agents (e.g. the middle one in the above figure) can be specified indirectly by providing "message handlers" written as Coq functions. Because Coq is a pure functional language and has no IO facilities, we provide a ROS shim which handles sending and receiving of messages for such Coq programs. Given a received message as input, message handlers compute messages that are supposed to be sent by the shim in response. They can also request the shim to send some messages at a later time. For example, to get a robot to turn by a right angle, one can send a message requesting a positive angular velocity (say 1 rad/s) and send another message requesting an angular velocity of 0 after time $\frac{\pi}{2}$s. While reasoning about the behavior of the system, we assume that the actual time a message is sent is not too different from the requested time.

Clearly robotics programs need to compute with real numbers. In CoRN, real numbers (e.g. π) are represented as Coq's functional programs that can be used to compute arbitrarily close rational (Q) approximations to the represented real number. Most operations on such reals are *exact*, e.g. Field operations, trigonometric functions, integral, etc. Some operations such as equality test are undecidable, hence only approximate. However, the error in such approximations can be made *arbitrarily* small (see Sect. 4.2). We prove a parametric upper bound on how far the robot will be from the position requested by the external agent. The parameters are bounds on physical imperfections, above mentioned computational errors, variations in message-delivery timings, etc. Using our shim, we ran our Coq program on an actual robot. We provide measurements over several runs and videos of the system in action.

Section 2 describes how to specify a physical model in ROSCoq. Section 3 describes the semantics of events and message delivery. Section 4 describes the semantics of agents. Section 5 describes some proof techniques for holistic reasoning about a CPS and the properties proven about our running example. It ends with a description of our experiments. Finally, we discuss related work and conclude. ROSCoq sources and more details are available at the companion website [17].

2 Physics

One of the first steps in developing a CPS using ROSCoq is to accurately specify its physical model. It describes how all of the relevant physical quantities in the system evolve over time. In our running example, these include the position and orientation of the robot and their derivatives. Using dependent types, one can

also include the constraints between the physical quantities, e.g. the constraint that velocity is the derivative of position w.r.t. time. Other examples include physical laws such as Newton's laws of motion, laws of thermodynamics. We use 2 of the 3 versions of constructive reals implemented in CoRN. In our programs which are supposed to be executed, we use the faster implementation (CR), while we use the slower (IR) for reasoning. CoRN provides field and order isomorphisms between these 2 versions. To avoid confusion, we use the notation $\mathbb{R}$ for both versions. However, clicking at colored text often jumps to its definition, either in this document or in external web pages.

Time is defined as non-negative reals, where 0 denotes the time when the system starts. For each relevant physical quantity, the physical model determines how it evolves over time. This can be represented as a member (say f) of the function type Time $\to \mathbb{R}$. The intended meaning is that for time t, $f\ t$ denotes the value of the physical quantity at time t. However, physical processes are usually continuous, at least at the scale where classical physics is applicable. For example, a car does not suddenly disappear at some time and appear miles apart at the exact same time. See [9] for a detailed discussion of the importance of continuity in physics. So, we choose to represent evolution of physical quantities as continuous functions. The type TContR is similar to the function type Time $\to$ $\mathbb{R}$, except that it additionally requires that its members be continuous functions. We have proved that TContR is an instance of the Ring typeclass [20], where ring operations on TContR are pointwise versions of the corresponding operations on real numbers. Apart from the proofs of the ring laws, this instance also involves proving that those ring operations result in continuous functions. As a result of this proof, one can use notations like $+$, $*$ on members of TContR, and the `ring` tactic of Coq can automate equational reasoning (about TContR expressions) that follows just from ring laws.

Using records, which are just syntactic sugars for dependent pairs, one can model multiple physical quantities and also the associated physical laws. The record type below defines the physical model in our running example. It represents how the physical state of an iCreate robot evolves over time.

```
Record iCreate : Type := {
  position : Cart2D TContR;
  theta : TContR;      linVel : TContR;      omega : TContR;

  derivRot : isDerivativeOf omega theta;
  derivX : isDerivativeOf (linVel * ( FCos theta)) (X position);
  derivY : isDerivativeOf (linVel * ( FSin theta)) (Y position);

  init1: ({X position} 0) ≡ 0 ∧ ({Y position} 0) ≡ 0;
  init2: ({theta} 0) ≡ 0 ∧ ({linVel} 0) ≡ 0 ∧ ({omega} 0) ≡ 0
}.
```

For any type, A, the type Cart2D A is isomorphic to the product type $A \times A$. X and Y are the corresponding projection functions. The type Polar2D A is similar, except that rad and θ are the projection functions. So, the first field (position) of the record type (iCreate) above is essentially a pair of continuous

functions, modeling the evolution of X and Y coordinates over time, respectively. The next line defines 3 fields which respectively model the orientation, linear velocity and angular velocity. The types of remaining fields depend on one or more of the first 4 fields. This dependence is used to capture constraints on the first four fields. The last 2 fields specify the initial conditions. The 3 fields in the middle characterize the derivatives of position and orientation of the robot. The first of those (derivRot) is a constructive proof/evidence that omega is the derivative of theta. The definition of the relation isDerivativeOf is based on CoRN's constructive notion of a derivative, which in turn is based on [4]. The next two are slightly more complicated and involve some trigonometry. FCos denotes the pointwise cosine function of type TContR → TContR. So, FCos theta is a function describing how the cosine of the robot's orientation (theta) evolves over time. Recall that here ∗ represents pointwise multiplication of functions. derivX and derivY imply that the linear motion of the robot is constrained to be along the instantaneous orientation of the robot (as defined by theta).

Our definition of a CPS is parametrized by an arbitrary type which is supposed to represent the physical model of the system. In the case of our running example, that type is (iCreate) (defined above). In the future, we plan to consider applications of our framework to systems of multiple robots. For a system of 2 iCreate robots, one could use the type (iCreate) × (iCreate) to represent the physical model. In Sect. 4, we will see that the semantics of hardware agents of a CPS is specified partly in terms of the physical model of the CPS.

3 Events and Message Delivery

As mentioned in Sect. 1, CPSs such as ensembles of robots can be thought of as distributed systems where agents might have sensing and/or actuation capabilities. The Logic of Events (LoE) framework has already been successfully used to reason about complicated distributed systems like fault-tolerant replicated databases [19]. It is based on seminal work by Lamport and formalizes the notion of message sequence diagrams which are often used in reasoning about the behavior of distributed systems. A distributed system (also a CPS) can be thought of as a collection of agents (components) that communicate via message passing. This is true at several levels of abstraction. In a collection of robots collaborating on a task (e.g. [5]), each robot can be considered as an agent. Moreover, when one looks inside one of those robots, one sees another CPS where the agents are components like software controllers, sensors and actuators. As mentioned before, the Robot Operating System (ROS) [15] presents a unified interface to robots where the subcomponents of even a single robot (e.g. sensors, actuators, controller) are represented as agents (a.k.a. nodes in ROS) that communicate with other agents using message passing. In a message sequence diagram such as the one in Sect. 1, agents are usually represented as vertical lines where the downward direction denotes passing of time. In ROSCoq, one specifies the collection of agents by an arbitrary type (say *Loc*) with decidable equality.

The next and perhaps most central concept in LoE is that of an event. In a message sequence diagram, these are points in the vertical lines usually denoting receipt or sending of messages by an agent. The slant arrows denote flight of messages. We model events by defining an abstract type *Event* which has a bunch of operations, such as:

eLoc : *Event* → *Loc*; eMesg : *Event* → Message;
causedBy : *Event* → *Event* → `Prop`; causalWf : well_founded causedBy

For any event *ev*, eLoc *ev* denotes the agent associated with the event. For receive-events, this is the agent who received the message. For send-events, this is the agent who sent the message. eMesg *ev* is the associated Message. The relation causedBy captures the causal ordering on events. causalWf formalizes the assumption that causal order is well-founded [10]. It allows one to prove properties by induction on causal order.

So far, our definition of an event is a straightforward translation (to Coq) of the corresponding Nuprl definition [19]. For CPSs, we need to associate more information with events. Perhaps the most important of those is a physical (as opposed to logical) notion of time when events happen. For example, the software agent needs to send appropriate messages to the hardware agent before the robot collides with something. One needs to reason about the time needed for activities like sensing, message delivery, computation to happen. So, we add the following operation:

eTime : *Event* → QTime;
globalCausal : ∀ (*e1 e2* : *Event*), causedBy *e1 e2* → (eTime *e1* < eTime *e2*)

For any event *ev*, eTime *ev* denotes the physical (Newtonian) time when it happened. QTime is a type of non-negative rational numbers where 0 represents the time when the system was started[3]. Note that this value of time is merely used for reasoning about the behavior of the system. As we will see later, a software controller cannot use it. This is because there is no way to know the exact time when an event, e.g. receipt of a message happened. For that, one would have to exactly synchronize clocks, which is impossible in general. One could implement provably correct approximate time-synchronization in our framework and then let the software controllers access an approximately correct value of the time when an event happened.

3.1 Message Delivery

Our message delivery semantics formalizes the publish-subscribe pattern used in ROS. The Coq definition of each agent includes a list of topics to which the agent publishes and a list of topics to which it subscribes. In ROSCoq, one can specify the collection of topics by an arbitrary type (say *Topic*) with decidable equality. In addition, one has to specify a function (say *topicType*) of type *Topic* → `Type`, that specifies the payload type of each topic. The type of messages can then be defined as follows:

[3] Section 4.1 explains the difference between Time and QTime.

```
Definition Message : Type := {tp : Topic × (topicType tp)} × Header .
```

A message is essentially a 3-tuple containing a topic (tp), a payload corresponding to tp, and a header. Currently the header of a message only has one field (delay) which can be used by software agents (Sect. 4.2) to request the shim to send a message at a later time. For our running example, we use 2 topics:

```
Definition topicType (t : Topic)
: Type := match t with
| VELOCITY ⇒ Polar2D Q
| TARGETPOS ⇒ Cart2D Q
end.
```

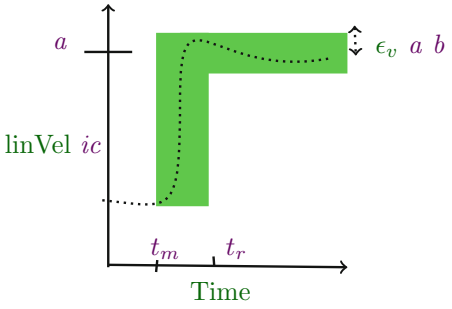
The topic TARGETPOS is used by the external agent (see Figure in Sect. 1) to send the cartesian coordinates of the target position (relative to the robot's current position) to the software agent. The topic VELOCITY is used by the software agent to send the linear and angular velocity commands to the robot hardware agent. One also provides a ternary relation to specify acceptable message delivery times between any two locations. Finally, we assume that message delivery is ordered.

4 Semantics of Agents

For verification of distributed systems [19], one assumes that each agent is running a functional reactive program. These programs indirectly specify a property about the sequence of events at an agent, namely it should be one that the program could generate. In a CPS, there usually are agents which represent hardware components (along with their ROS drivers) like sensors and actuators. Often, informal specifications about their behavior is available, not their internal design or firmware. Hence, one needs to axiomatically specify the observable behavior (sequence of events) of such devices. Moreover, these hardware devices often depend on (e.g. sensors), or influence (e.g. actuators) the evolution of some physical quantities. A specification of their behavior needs to talk about how the associated physical quantities evolve over time. Hence, an appropriate way of specifying the behavior of agents is to specify them as a *relation* between the physical model (how the physical quantities evolve over time) and the sequence of messages associated with the agent. As we will see below, for hardware agents one can directly specify that relation. For software agents, this relation would only have a vacuous dependence on the physical model and can be specified indirectly as a Coq program, which is often more succinct (Sect. 4.2).

4.1 Hardware Agents

For our running example, the physical model is specified by the type (iCreate) (Sect. 2). The type $\mathbb{N} \to$ (option *Event*) can be used to represent a possibly finite sequence of events. So, the specification of the behavior of the hardware agent is a relation (HwAgent) of the following type: (iCreate) $\to$ ($\mathbb{N} \to$ option *Event*) $\to$ Prop.



We will first explain it pictorially and then show the actual Coq definition. iCreate is primarily an actuation device and this relation asserts how the angular and linear velocity (see omega and linVel in the definition of (iCreate) above) of the robot changes in response to the received messages. It is quite close to informal manuals[4]. The iCreate hardware driver only receives messages on the topic VELOCITY. It reacts to such messages by adjusting the speed of the two motors (one on each side) so the robot's linear and angular velocities are close to the requested values. The figure above illustrates how the linear velocity of an iCreate is supposed to change in response to a message requesting a linear velocity a and angular velocity b. t_m denotes the time when the message was received. HwAgent asserts that there must exist a time t_r by which the linear velocity of the robot becomes close to a. The parameter *reacTime* is an upper bound on t_r - t_m. ϵ_v and ϵ_ω are parameters modeling the actuation accuracy. After time t_r, the linear velocity of the robot remains at least ϵ_v a b close to the a. Similarly (not shown in the figure) the angular velocity remains at least ϵ_ω a b close to b. The only assumption we make about the ϵ is that ϵ_v 0 0 and ϵ_ω 0 0 are 0, i.e. the robot complies exactly (after a certain amount of reaction time) when asked to both stop turning and stop moving forward. In particular, we don't assume that ϵ_ω a 0 is 0. When the robot is asked to move forward at a m/s and not turn at all, it may actually turn a bit. For a robot to move in a perfect straight line, one will likely have to make sure that the size of the two wheels are *exactly* the same, the two motors are getting *exactly* the same amount of current and so on. A consequence of our realistic assumptions is that some integrals become more complicated to reason about. For example, unlike in the case for perfect linear motion, the angle in the derivative of position cannot be treated as a constant. Here is the definition of HwAgent (mentioned above), which captures the above pictorial intuition:

```
Definition HwAgent (ic: iCreate) (evs : nat → option Event): Prop :=
onlyRecvEvts evs ∧ ∀ t: QTime,
  let (lastCmd, t_m ) := latestVelPayloadAndTime evs t in
  let a : Q := rad (lastCmd) in
  let b : Q := θ (lastCmd) in ∃ t_r : QTime, (t_m ≤ t_r ≤ t_m + reacTime)
    ∧ (∀ t' : QTime, (t_m ≤ t' ≤ t_r)
        → ( Min ({linVel ic} t_m) (a - ε_v a b)
            ≤ {linVel ic} t' ≤ Max ({linVel ic} t_m) (a+ ε_v a b)))
    ∧ (∀ t' : QTime, (t_r ≤ t' ≤ t) → |{linVel ic} t' - a | ≤ ε_v a b )
```

The function latestVelPayloadAndTime searches the sequence of events *evs* to find the latest message of VELOCITY topic received before time t. We assume

[4] http://pharos.ece.utexas.edu/wiki/index.php/Writing_A_Simple_Node_that_Moves_the_iRobot_Create_Robot\#Talking_on_Topic_cmd_vel.

that there is a positive lower-bound on the time-gap between any two events at an agent. Hence, one only needs to search a finite prefix of the sequence *evs* to find that event. It returns the payload of that message and the time the corresponding event occurred (e.g. t_m in the figure). If there is no such message, it returns the default payload with 0 as the velocities and 0 as the event time. The last conjunct above captures the part in the above figure after time t_r. The 2^{nd} last conjunct captures the part before t_r where the motors are transitioning to the new velocity. There are 2 more conjuncts not shown above. These express similar properties about angular velocity (omega *ic*), b and $\epsilon_\omega\ a\ b$.

Because the semantics of a hardware agent is specified as a relation between the physical model and the sequence of events at the agent, it is equally easy to express the specification of sensing devices where typically the physical model determines the sequence of events. [17] contains a specification of a proximity sensor. Although the external agent in our running example is not exactly a hardware agent, we specify its behavior axiomatically. We assume that there is only one event in its sequence, and that event is a send event on the topic TARGETPOS.

We conclude this subsection with an explanation of some differences between Time and QTime. The former represents non-negative real numbers, while the latter represents non-negative rational numbers. Clearly, there is an injection from QTime to Time. We have declared this injection as an implicit coercion, so one can use a QTime where a Time is expected. Because CoRN's theory of differential calculus is defined for functions over real numbers, we can directly use them for functions over real-valued time (i.e. TContR). However, QTime is often easier to use because comparison relations (equality, less than etc.) on rationals are decidable, unlike on real numbers. For example, if the time of events (eTime) were represented by Time, one could not implement the function latestVelPayloadAndTime mentioned above. Because members of TContR are *continuous* functions over real-valued time, they are totally defined merely by their value on rational numbers, i.e. QTime. For example, the specification above only bounds velocities of the robot at rational values of time. However, it is easy to derive the same bound for all other values of time.

4.2 Software Agents

As mentioned before in this section, the behavior of software agents can be specified indirectly by just specifying the Coq program that is being run by the agent. Following [23], these programs are message handlers which can also maintain some state of an arbitrary type. A software agent which maintains state of type S can be specified as a Coq function of the following type: $S \rightarrow \text{Message} \rightarrow (S \times \mathsf{list}\ \text{Message})$. Given the current state and a received message, a message handler computes the next state and a list of messages that are supposed to be sent in response. We provide a ROSJava[5] shim which handles sending and receiving of messages for the above pure functions. It communicates with the Coq

[5] http://wiki.ros.org/rosjava.

toplevel (coqtop) to invoke message handlers. It also converts received messages to Coq format and converts the messages to be sent to ROSJava format. However, the state is entirely maintained in Coq, i.e. never converted to Java. We define SwSemantics, a specification of how the shim is supposed to respond to received messages. For a message handler, it defines the behavior of the corresponding software (Sw) agent, essentially as a property of the sequence of (send/receive) events at the agent. As mentioned before, the semantic relation of a software agent has vacuous dependence on the physical model. A software agent does not directly depend on or influence physical quantities of a CPS. It does so indirectly by communicating with hardware agents like the one described in the previous subsection.

The definition of our shim (in Java) and SwSemantics (in Coq) can be found at [17]. Here, we explain some key aspects. SwSemantics asserts that whenever a message m is received (at a receive event), the message handler (in Coq) is used to compute the list of messages (say l) that are supposed to be sent. There will be $|l|$ send events which correspond to sending these messages one by one. Let s_i be the time the i^{th} of these send events happened. Recall from Sect. 3.1 that the header of messages contain a delay field. Let d_i be the value of the delay field of the i^{th} message in the list l. Let t be the time the computation of l finished. The shim ensures that s_0 is close to $t + d_0$. It also ensures that $\forall$ appropriate i, s_{i+1} is close to $d_{i+1} + s_i$. The current state is updated with the new state computed along with l. SwSemantics also asserts that there are no other send events; each send event must be associated to a receive event in the manner explained above.

In our running example, the software agent receives a target position for the robot on the topic TARGETPOS and sends velocity-control messages to the motor on the topic VELOCITY. Recall from Sect. 3.1 that the payload type for the former and latter topics are Cart2D Q and Polar2D Q respectively. So the software agent reacts to data of the former type and produces data of the latter type. Its program can be represented as the following pure function:

```
Definition robotPureProgam (target : Cart2D Q) : list (Q × Polar2D Q) :=
  let polarTarget : Polar2D ℝ := Cart2Polar target in
  let rotDuration : ℝ := | θ polarTarget | / rotspeed in
  let translDuration : ℝ := (rad polarTarget) / speed in
  [ (0,{| rad:= 0 ; θ := ( polarθSign target ) * rotspeed |})
      ; ( tapprox rotDuration delRes delEps , {| rad := 0 ; θ := 0 |})
      ; (delay , {| rad :=  speed ; θ := 0 |})
      ; ( tapprox translDuration delRes delEps , {| rad := 0 ; θ := 0 |}) ].
```

For any type A and a and b of type A, {| X := a ; Y := b |} denotes a member of type Cart2D A. {| rad := a ; θ := b |} denotes a member of type Polar2D A. The program produces a list of 4 pairs, each corresponding to one of the 4 messages that the software agent will send to the hardware agent (see Figure in Sect. 1). One can compose this program with ROSCoq utility functions to lift it to a message handler. The first component of each pair denotes the delay field of the message's header. The second component corresponds to the payload of the message. Recall that a payload {| rad := a ; θ := b |} represents a request

to set the linear velocity to a and the angular velocity to b. In the above program, *polarTarget* represents the result of converting the input to polar coordinates. Note that even though the coordinates in *target* are rational numbers, those in *polarTarget* are real numbers. For example, converting {| X := 1 ; Y := 1 |} corresponds to irrational polar coordinates: {| rad := $\sqrt{2}$; θ := $\frac{\pi}{4}$ |}.

The program first instructs the robot to turn so that its orientation is close to θ *polarTarget*, i.e., in the direction of *target*. *speed*, *rotspeed*, *delEps*, *delRes*, *delay* are parameters in the program. These are arbitrary positive rationals. The robot will turn at speed *rotspeed*, but it can turn in either direction : clockwise or counter-clockwise, depending on the sign of θ *polarTarget*. However, the problem of finding the sign of a real number is undecidable in general. Fortunately, because θ *polarTarget* was computed from rational coordinates (*target*), one can look at *target* and indirectly determine what the sign of θ *polarTarget* would be. We have proved that *polarθSign* does exactly that. *polarθSign target* will either be $+1$ or -1 (in the first message).

The 2^{nd} message which requests the robot to stop (turning) should ideally be sent after a delay of *rotDuration*, which is defined above as | θ *polarTarget* | / *rotspeed*. It is a real number because θ *polarTarget* is so. However our Java shim currently uses *java.util.Timer*[6] and only accepts delay requests of integral number of milliseconds[7]. It might be possible to use a better hardware/shim to accept delay requests of integral number of microseconds or nanoseconds. So we use an arbitrary parameter *delRes* which is a positive integer such that $\frac{1}{delRes}$ represents the resolution of delay provided by the shim. For our current shim, one will instantiate *delRes* to 1000. So, we should approximate *rotDuration* by the closest rational number whose denominator is *delRes*. In classical mathematics, one can prove that there "exists" a rational number that is at most $\frac{1}{2*delRes}$ away from *rotDuration*. However, *finding* such a rational number is an undecidable problem in general. Fortunately, one can arbitrarily minimize the suboptimality in this step. We have proved that for *any* positive rational number *delEps*, tapprox *rotDuration delRes delEps* is a rational number whose denominator is *delEps* and is at most $\frac{1+2*delEps}{2*delRes}$ (denoted as R2QPrec) away from *rotDuration*.

tapprox was easy to define because CoRN's real numbers of the type CR are functional programs which approximate the represented real number to arbitrarily close rationals. Note however that cartesian to polar conversion was exact. Unlike with floating points, most operations on real numbers are exact : field operations, trigonometric functions, integrals, etc. One does not have to worry about errors at each step of computation. Instead, one can directly specify the desired accuracy for the final discrete result. So we think constructive reals are ideal for robotic programs written with the intent of rigorous verification.

The 3^{rd} message sets linear velocity to *speed*. The parameter *delay* is the delay value for this message. We assume *delay* is large enough (w.r.t other parameters, e.g. *reacTime*) to ensure that motors have fully stopped in response to the previous message by the time this message arrives. The final message asks the robot to stop. Again it is sent after a nearly right amount of delay.

[6] http://docs.oracle.com/javase/7/docs/api/java/util/Timer.html.

[7] Also, recall that the shim is only required to approximately respect these requests.

5 Reasoning About the System

After all the agents of a CPS have been specified, one can reason about how the overall system will behave. For local reasoning about an agent's behavior, one can use natural induction on its sequence of events. For global behavior, one can use induction on the causal order of messages. In our running example, we are interested in how close the robot will be to the target position. In the previous section we already saw that there might be some error in approximating real numbers to certain rational values of time which the shim can deal with. However, that was just one source among myriad other sources of errors : messages cannot be delivered at exact times, actuation devices are not perfect (infinitely precise), and so on. Our goal is to derive parametric bounds on how far the robot can be from the ideal target position, in terms of bounds on the above error sources. Below, we consider an arbitrary run of the system. The external agent asks the robot to go to some position *target* of type Cart2D Q. *ic* of type (iCreate) denotes how the physical quantities evolve in this run.

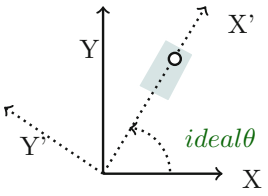
The first step is to prove that the sequence of events at each agent looks exactly like the figure in Sect. 1. In particular, we prove that there are exactly 4 events at the hardware agent and those events correspond to the four messages (in order) computed by the program described in the previous section. These proofs mostly involve using the properties about topic-subscriptions and guarantees provided by the messaging layer, such as guaranteed and ordered delivery of messages. We use mt0, ..., mt3 to refer respectively to the time of occurrences of those 4 events. The remaining proofs mostly involve using the specification of motor to characterize the position, orientation and velocities of the robot at each of those times. The specification of motor provides us bounds on velocities. We then use CoRN's lemmas on differential calculus, such as the FTC to characterize the position and orientation of the robots.

Because we assumed that ϵ_v 0 0 and ϵ_ω 0 0 are both 0 (Sect. 4.1), and initial velocities (linear and angular) are 0, the velocities will remain *exactly* 0 till mt0. So the position and orientation of the robot at mt0 is exactly the same as that in the initial state (initial conditions are specified in the definition of iCreate). At mt0, the robot receives a message requesting a non-zero angular velocity (say w). Recall that w is either *rotspeed* or - *rotspeed*. Between mt0 and mt1, the robot turns towards the target position. At mt1, it receives a message to stop, however it might take some time to totally stop. At mt2, it receives a message to start moving forward. Ideally, it should be oriented towards the target position by mt2. However, that might not be the case because of several sources of imperfections. The following lemma characterizes how imperfect the orientation of the robot can be at mt2.

```
Definition idealθ : ℝ := θ (Cart2Polar target ).
Definition θErrTurn : ℝ := rotspeed * ( timeErr + 2 * reacTime)
  + (ε_ω 0 w ) * (timeErr + reacTime) +(( ε_ω 0 w) / rotspeed) * | idealθ | .
Lemma ThetaAtEV2 : | {theta ic} mt2 - idealθ | ≤ θErrTurn.
```

Because Sin and Cos are periodic, there are several ways to define θ (Cart2Polar *target*). Our choice enabled us to prove that it is in the range $[-\pi, \pi]$. It minimizes the turning that the robot has to do (vs., e.g. $[0, 2\pi]$). It also enables us to replace | *idealθ* | by π in the above upper bound. The three terms in the definition *θErrTurn* correspond to errors that are respectively proportional, independent and inversely proportional to *rotspeed*. Recall (Sect. 4.1) that *reacTime* is the upper bound on the amount of time the robot takes to attain the requested velocity. timeErr has been proved to be an upper bound on the error of the value mt1 - mt0 w.r.t. its ideal value. It is an addition of terms bounding the inaccuracy of sending times, variance of message delivery times, R2QPrec which bounds the inaccuracy introduced when we approximated the ideal real-valued delay by a rational value (Sect. 4.2). A higher value of *rotspeed* means that the error in the duration of turn will lead to more errors in the final angle. A lower value of *rotspeed* increases the duration for which the robot has to turn, thus accumulating more errors because of imperfect actuation of angular velocity (as modeled by ϵ_w 0 w). However, if ϵ_w 0 w is directly proportional to the absolute value of w, which is *rotspeed*, the division in the last term will cancel out. In such a case, a lower value of *rotspeed* will always result in a lower upper bound on turn error.

From mt2 to mt3, the robot will move towards the target. To analyse this motion, we find it convenient to rotate the axis so that the new X axis (X') points towards the target position (shown as a circle in the RHS figure). We characterize the derivative of the position of the robot in the rotated coordinate frame:




```
Definition Y'Deriv : TContR :=
  (linVel ic) * (FSin (theta ic - FConst idealθ)).
Definition X'Deriv := (linVel ic) * (FCos (theta ic - FConst idealθ)).
```

The advantage of rotating axes is that unlike theta *ic* which could be any value (depending on *target*), (theta *ic* - FConst *idealθ*) is a small angle: $\forall\ t$, mt2 $\leq t \leq$ mt3 $\rightarrow$ |{theta *ic*} t - *idealθ*| $\leq$ *θErrTrans* + *θErrTurn*.

As explained above, *θErrTurn* is a bound on the error (w.r.t. *idealθ*), at mt2. Even though the robot is supposed to move straight towards the goal from time mt2 to mt3, it might turn a little bit due to imperfect actuation (as discussed in Sect. 4.1). We proved that *θErrTrans* is an upper bound on that. *θErrTrans* is proportional to mt3 - mt2, which in turn is proportional to the distance of the target position from the origin. For the remaining proofs, we assume *θErrTrans* + *θErrTurn* $\leq \frac{\pi}{2}$, which is a reasonable assumption unless the target position is too far away or the actuation is very imprecise. In other words, we are assuming that there cannot be more than a difference of 90 degrees between the direction the robot thinks it is going and the actual direction. For robots that are supposed to move for prolonged periods of time, one usually needs a localization mechanism such as a GPS and/or a compass. In the future, we plan to consider such closed-loop setups by adding another hardware agent

in our CPS which will periodically send much more accurate estimates of the robot's position and orientation as messages to the software agent.

Using the above assumption, it is easy to bound the derivatives of the robot's position in the rotated coordinate frame. For example, between times mt2 and mt3, the value of (FSin (theta *ic* − FConst *ideal*θ)) will be bounded above by the constant Sin ($\theta ErrTrans$ + $\theta ErrTurn$). We prove that at mt3, the robot will be inside a rectangle which is aligned to the rotated axes. In the above figure, such a rectangle is shown in gray. Recall that mt3 is the final event in the system, where the robot receives a message requesting it to stop. The following defines the upper bound we proved on the distance of the X' aligned sides of the rectangle from the X' axis. In other words, it is a bound on the Y' coordinate of the robot in the rotated coordinate frame. Ideally, this value should be zero.

```
Definition ErrY': ℝ := (ε_v 0 w) * ( reacTime + Ev01TimeGapUB)
+ (Sin (θErrTrans + θErrTurn)) * ( | target | + speed*timeErr
                                  + Ev23TimeGapUB * (ε_v speed 0) ) .
```

The first line of the above definition corresponds to the error accrued in the position while turning (between mt0 and a little after mt1 when turning totally stops). The second and third lines denotes the error accrued after mt2 when the robot moves towards the target position. Similarly, we proved bounds on the distance of the Y' aligned sides from the Y' axis (see [17]).

We also considered the case of a *hypothetical* train traveling back and forth repeatedly between two stations. This CPS has 3 hardware agents : a proximity sensor at each end of the train and a motor at the base for 1D motion. The software controller uses messages generated by the proximity sensor to reverse the direction of motion when it comes close to an endpoint. We proved that it will never collide with an endpoint [17]. We haven't physically implemented it.

5.1 Experiments

Using our shim, we were able to use the Coq program in Sect. 4.2 to actually drive an iCreate robot to the position requested by a human via a GUI. While a detailed estimation of parameters in the model of hardware, message delivery, etc. is beyond the scope of this paper, we did some experiments to make sure that the robot is in the right ball park. The table above shows some measurements (in meters) from the experiments.

Target		Actual		Video link
X	Y	X	Y	
−1	1	−1.06	0.94	vid1
−1	−1	−1.02	−0.99	vid2
1	1	1.05	0.94	vid3

6 Related Work

Hybrid automata [2] is one of the earliest formalisms to simultaneously model and reason about both the cyber (usually discrete dynamics) and physical (usually continuous dynamics) components of a CPS. Several tools have been developed for approximate reachability analysis, especially for certain sub-classes of

hybrid automata (see [1] for a survey). However, hybrid automata provide little structure to implement complicated CPSs in a modular way. Also for CPSs with several communicating agents, it is rather non-trivial to come up with a hybrid automata model which accounts for all possible interactions in such distributed systems. In ROSCoq, we independently specify the agents of a distributed CPS and explicitly reason about all possible interactions.

The KeYmaera [14] tool takes a step towards more structural descriptions of CPSs. It has a non-deterministic imperative language to specify hybrid programs. It also comes with a dynamic-logic style proof theory for reasoning about such programs [13]. Unlike ROSCoq, the semantics of KeYmaera's programming language pretends that one can exactly compare two real numbers, which is impossible in general. When one uses floating point numbers to implement such programs, the numerical errors can add up and cause the system to violate the formally proven properties [11]. In contrast, the use of constructive reals forces us to explicitly account for inexactness of certain operations (like comparison) on real numbers and hence there is one less potential cause of runtime errors. In [21], they consider 2D dynamics similar to ours. They don't consider the possibility of the robot turning a little when asked to go straight. Finally, the semantics of their system assumes that all the robots are executing a synchronized control loop. Our asynchronous message passing based model is more realistic for distributed robotic systems.

Unlike the above tools, our focus is on correct-by-construction, i.e. we intend to prove properties of the actual software controller and not a simplified model of it. Some tools [16,18] automatically synthesize robot-controllers from high-level LTL specifications. However, these fully automatic approaches do not yet scale up to complicated robotic systems. Also, the specifications of these controllers are at a very high level (they discretize the continuous space) and do not yet account for imperfections in sensing, actuation, message delivery, etc.

Unlike the above formalisms, in ROSCoq one uses Coq's rich programming language to specify their hybrid programs and its powerful higher order logic to succinctly express the desirable properties. Coq's dependent types allow one to reuse code and enforce modularity by building interfaces that seamlessly specify not only the supported operations but also the logical properties of the operations. To trust our proofs, one only needs to trust Coq's type checker. Typical reasoning in KeYmaera relies on quantifier elimination procedures implemented using Mathematica, a huge tool with several known inconsistencies [6]. Our framework did not require adding any axiom to Coq's core theory. This is mainly because CoRN's real numbers are actual computable functions of Coq, unlike the axiomatic theory of reals in Coq's standard library. Interactive theorem provers have been previously used to verify certain aspects of hybrid systems [7,12]. Like ROSCoq, [7] uses constructive reals and accounts for numerical errors. However, it only supports reasoning about hybrid systems expressed as hybrid automata. [12] is primarily concerned about checking absence of collisions in completely specified flight trajectories.

7 Conclusion and Future Work

We presented a Coq framework for developing certified robotic systems. It extends the LoE framework to enable holistic reasoning about both the cyber and physical aspects of such systems. We showed that the constructive theory of analysis originally developed by Bishop and later made efficient in the CoRN project is powerful enough for reasoning about physical aspects of practical systems. Constructivity is a significant advantage here because the real numbers in this theory have a well defined computational meaning, which we exploit in our robot programs. Our reasoning is very detailed as it considers physical imperfections and computational imperfections while computing with real numbers.

We plan to use our framework to certify more complicated systems involving collaboration between several robots [5]. Also, we plan to develop tactics to automate as much of the reasoning as possible. We thank Jean-Baptiste Jeannin, Mark Bickford, Vincent Rahli, David Bindel and Gunjan Aggarwal for helpful discussions, Bas Spitters and Robbert Krebbers for help with CoRN, and Liran Gazit for providing the robot used in the experiments.

References

1. Alur, R.: Formal verification of hybrid systems. In: EMSOFT, pp. 273–278. IEEE (2011)
2. Alur, R., Courcoubetis, C., Henzinger, T.A., Ho, P.-H.: Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In: Grossman, R.L., Nerode, A., Ravn, A.P., Rischel, H. (eds.) HS 1993. LNCS, vol. 736, pp. 209–229. Springer, Heidelberg (1993)
3. Bickford, M., Constable, R.L., Eaton, R., Guaspari, D., Rahli V.: Introduction to EventML (2012). www.nuprl.org/software/eventml/IntroductionToEventML.pdf
4. Bishop, E., Bridges, D.: Constructive Analysis, p. 490. Springer Science and Business Media, New york (1985)
5. Dogar, M., Knepper, R.A., Spielberg, A., Choi, C., Christensen, H.I., Rus, D.: Towards coordinated precision assembly with robot teams. In: ISER (2014)
6. Duráan, A.J., Péerez, M., Varona, J.L.: the misfortunes of a trio of mathematicians using computer algebra systems. Can we trust in them? In: AMS Notices 61.10, p. 1249, November 1 2014
7. Geuvers, H., Koprowski, A., Synek, D., van der Weegen, E.: Automated machine-checked hybrid system safety proofs. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 259–274. Springer, Heidelberg (2010)
8. Krebbers, R., Spitters, B.: Type classes for efficient exact real arithmetic in Coq. In: LMCS 9.1, February 14 2013
9. Lamport, L.: Buridan's principle. In: Foundations of Physics 42.8, pp. 1056–1066, August 1 2012
10. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7), 558–565 (1978)
11. Mitsch, S., Platzer, A.: ModelPlex: Verified runtime validation of verified cyber-physical system models. In: Bonakdarpour, B., Smolka, S.A. (eds.) RV 2014. LNCS, vol. 8734, pp. 199–214. Springer, Heidelberg (2014)

12. Narkawicz, A., Munoz, C.A.: Formal verification of con ict detection algorithms for arbitrary trajectories. In: Reliable Computing, this issue (2012)
13. Platzer, A.: Logics of dynamical systems. In: LICS 2012, pp. 13–24 (2012)
14. Platzer, A., Quesel, J.-D.: KeYmaera: A hybrid theorem prover for hybrid systems (system description). In: AR, pp. 171–178. Springer (2008)
15. Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., Ng, A.Y.: ROS: an open-source robot operating system. In: ICRA Workshop on Open Source Software. vol. 3, p. 5 (2009)
16. Raman, V. Kress-Gazit, H.: Synthesis for multi-robot controllers with inter- leaved motion. In: ICRA, pp. 4316–4321, May 2014
17. ROSCoq online reference. http://www.cs.cornell.edu/~aa755/ROSCoq
18. Sarid, S., Xu, B., Kress-Gazit, H.: Guaranteeing high-level behaviors while exploring partially known maps. In: RSS, p. 377, Sydney July 2012
19. Schiper, N., Rahli, V., Renesse, R.V., Bickford, M., Constable, R.L.: Developing correctly replicated databases using formal tools. In: DSN, pp. 395–406. IEEE (2014)
20. Spitters, B., Van Der Weegen, E.: Type classes for mathematics in type theory. MSCS **21**(4), 795–825 (2011)
21. Mitsch, S., Ghorbal, K., Platzer, A.: On provably safe obstacle avoidance for autonomous robotic ground vehicles. In: RSS (2013)
22. Talcott, C.: Cyber-physical systems and events. In: Wirsing, M., Banâtre, J.-P., Hölzl, M., Rauschmayer, A. (eds.) Soft-Ware Intensive Systems. LNCS, vol. 5380, pp. 101–115. Springer, Heidelberg (2008)
23. Wilcox, J.R., Woos, D., Panchekha, P., Tatlock, Z., Wang, X., Ernst, M.D., Anderson, T.: Verdi: a framework for implementing and formally verifying distributed systems. In: PLDI, ACM (2015)