A Framework for Verifying the Collision Freeness of Collaborative Robots (Work in Progress)

Artur Graczyk Marialena Hadjikosti Andrei Popescu
{apgraczyk1,mhadjicosti1,a.popescu}@sheffield.ac.uk

University of Sheffield, UK

Abstract. Collision avoidance is a major problem when robotic devices are being deployed to perform complex collaborative tasks. We present a vision for a framework that makes it convenient to program collaborative robots and to verify that their behaviour is collision-free. It consists of a domain-specific language that is shallowly embedded in the ROS (Robot Operating System) framework and a translation into a programming language that is deeply embedded in the Isabelle/HOL theorem prover.

1 Vision

Our research targets collaborative tasks involving the movement of robots (mobile platforms or robotic arms). Such tasks are naturally expressed as controlled sequences of commands requesting the movement of robots to specific locations where to perform specific activities.

Collisions between two robots can occur in several scenarios, including:

Scenario 1. Two robots must visit the same location during a task. E.g., one robot brings an item to a location, from where another robot picks it up.
Scenario 2. While moving between two locations, the trajectories of two robots intersect. E.g., two robots need to enter an enclosure through the same gate.
Scenario 3. The trajectory of one robot touches the location where another robot performs a task. E.g., the second robot does a repairing job at the gate through which the first robot must enter.

Each scenario above suggests a possible collision point in space, i.e., a point that each of the two robots need to visit. Communication with the robots and subsequent synchronization can avoid such collisions, for example:

- In Scenario 1, the program can ask the second robot to wait at a safe distance until the first robot has successfully delivered the object to the given location and has moved away from that location.
- In Scenario 2, one can implement a mutual exclusion protocol whereby the first robot who announces its intention to go through the gate makes any other robot who wants to enter, pause and wait.
- In Scenario 3, the robot needing to pass through the gate could wait for the repairer robot to finish; or one might implement a more involved protocol whereby, depending on the urgency of the task, the repairer robot could pause its activity and move aside to let the other robot pass.

What all the discussed scenarios have in common is that they are fairly abstract: we can understand the collision problems that they raise in terms of the coordinates of the targeted locations and (a conservative over-approximation of) the sizes of the robots. And the solutions we discussed for these problems are equally abstract; indeed, they can be expressed in a high-level robot command language that tells the robots *what* movements to make, and in which sequence and under which conditions—but does not indicate *how* they should move from one location to another. In short, under this abstract view, both the collision problems and their solutions refer to the *what* but not to the *how*. This abstract view enables the following fruitful analogy: *Robot Collisions* $\simeq$ *Data Races.*

Of course, for collisions we are not talking about the usual data races, which take place in the digital space, but about *races in the physical space*. However, the ideas are essentially the same: While in concurrency one requests that the executions of two processes (or threads) never enter certain critical sections of their code at the same time, with collisions one requests that, during a collaborative task, the trajectories of two robots are never crossing each other. Moreover, it is in principle possible for robot programs to "internalize" the information about collision, in that the avoidance of collision can be mapped to the execution of certain critical sections, just like in concurrency. This has an important methodological consequence: *Verification and analysis techniques from concurrency can be adapted to produce collision avoidance guarantees for collaborating robots.* In this ongoing work, we are bringing to fruition some of the consequences of this analogy with concurrent programming:

- developing a formal semantics of a robot programming language and a property specification language that regard collisions as a form of data races, and
- performing an encoding of collisions as actual data races whose absence can be proved using concurrency verification and analysis tools.

2 Achieving the Vision

Achieving this vision requires the following:

- choosing and adapting a robot programming environment (discussed in §2.1),
- setting up a formal verification framework (discussed in § 2.2), and
- building the infrastructure for protected navigation, including a path analysis tool for computing safe corridors (not detailed in this paper).

Importantly, it also requires connecting these three components, which must exchange data between each other in order to achieve overall *strong collision freeness guarantees covering both fundamental and incidental collision hazards.* This will happen through the following tools:

- a code generator connecting the verification framework with the robotic programming environment,
- a tool for automatically transporting information from within the verification framework to a navigation path analysis tool, and
- a tool for plugging the computed corridors into the ROS run-time environment, to change the way robots view the physical space for navigation purposes.

Of course, there will be some assumptions about the robot environment and the navigation system in order for the formally proved guarantees to hold; for example, it will be assumed that the navigation system keeps the robot within the limits of the computed corridors (which in turn relies on the correctness of certain navigation algorithms).

2.1 Robot Programming Environment

Our goal is to have a practical programming environment, in which we can easily program collaborative robotic tasks in a transparent fashion, suitable for verification. Because of the practicality desideratum, we have chosen the ROS system as implemented in Python via the *rospy* library—which is widely used by robot practitioners.

However, at its core the ROS framework employs a model of communication (based on topics and/or services) that is very versatile but somewhat bureaucratic. For this reason, we implemented, on top of ROS, an API for enabling more direct communication with the robotic devices (§2.1.1). The user of the API does not need to explicitly create, use or subscribe to topics or services. Rather, the creation of these entities happens behind the scenes, and the API allows one to issue direct movement commands to the robots, and send inquiries to them about their status.

The API thus abstracts away from the communication complexity, allowing one to focus on programming the movements of the robots, which makes the programs more amenable to collision-freeness verification. As it will be sketched in § 2.2, the functions of the API have been not only implemented in Python, but have also been given a formal semantics in our verification framework—for the purpose of connecting the programming and verification platforms. In addition to the API, the connection between the two platforms also requires an identification of a domain-specific language (DSL), covered by a formal semantics. Our DSL is a Turing-complete multi-threaded fragment of Python featuring calls to the API as atomic statements (§2.1.2).

2.1.1 ROS-based API The API consists of functions for initializing, moving and sending inquiries to the robots for various pieces of information. More precisely, we have the following functions, with the following behaviors:

- initRobot: Takes a robot ID and the coordinates of a desired initial position and initializes a robot at the given position.
- moveTo: Takes a robot ID and a target position and issues a command to the indicated robot for moving to the indicated position.
- getSuccessStatus: Takes a robot ID and sends an inquiry to the given robot on whether the last attempted action was completed successfully.
- getMovingStatus: Takes a robot ID and determines whether the robot is currently moving.
- getPosition: Takes a robot ID and returns the robot's current position.

These functions were chosen to have a simple and intuitive semantics, and to allow the programming of interesting examples.

2.1.2 Python/ROS-based DSL

Below is the syntax of our domain-specific language:

$$
\begin{array}{lll}
\mathsf{Lit} & ::= & \mathsf{IntLit} \mid \mathsf{RealLit} \mid \mathsf{BoolLit} \mid \mathsf{StringLit} \\
\mathsf{Op}_1 & ::= & - \mid \mathsf{not} \\
\mathsf{Op}_2 & ::= & + \mid - \mid * \mid \% \mid = \mid \mathsf{and} \mid \mathsf{or} \\
\mathsf{RobotID} & ::= & \mathsf{StringLit} \\
\mathsf{Exp} & ::= & \mathsf{Var} \mid \mathsf{Lit} \mid \mathsf{Op}_1\ \mathsf{Exp} \mid \mathsf{Exp}\ \mathsf{Op}_2\ \mathsf{Exp} \mid \\
 & & \mathsf{getSuccessStatus}\ (\mathsf{RobotID}) \mid \\
 & & \mathsf{getMovingStatus}\ (\mathsf{RobotID}) \mid \\
 & & \mathsf{getPosition}\ (\mathsf{RobotID})
\end{array}
$$

$$
\begin{array}{lll}
\mathsf{Coord} & ::= & (\mathsf{RealLit}, \mathsf{RealLit}) \quad \text{for now, two dimensions only} \\
\mathsf{ACom} & ::= & \mathsf{skip} \mid \mathsf{Var} = \mathsf{Exp}; \mid \mathsf{moveTo}\ (\mathsf{RobotID}, \mathsf{Coord}); \\
\mathsf{InitGV} & ::= & \mathsf{initGlobalVar}(\mathsf{Var}, \mathsf{Exp}) \\
\mathsf{RobotInfo} & ::= & \ldots \\
\mathsf{InitR} & ::= & \mathsf{initRobot}(\mathsf{RobotID}, \mathsf{RobotInfo}) \\
\mathsf{ISec} & ::= & \mathsf{InitGV}^*\ \ \mathsf{InitR}^* \\
\mathsf{Com} & ::= & \mathsf{ACom} \mid \\
 & & \mathsf{Com}\ \mathsf{Com} \mid \\
 & & \mathsf{if}\ (\mathsf{Exp})\ \{\mathsf{Com}\}\ \mathsf{else}\ \{\mathsf{Com}\} \mid \\
 & & \mathsf{while}\ (\mathsf{Exp})\ \{\mathsf{Com}\} \\
\mathsf{Thread} & ::= & \mathsf{ISec}\ \ \mathsf{Com} \\
\mathsf{Prog} & ::= & \mathsf{Thread}^*
\end{array}
$$

We have the usual real, integer, boolean and strings expressions (built from literals and operators—logical, arithmetical, etc.), as well as calls to our API observation functions (in blue), which take robot IDs parameters (string literals). Only well-typed programs are accepted, but we omit the obvious typing.

The atomic commands are skip (i.e., "do nothing"), assignments of expressions to variables, and calls to the robot moving function of our API. The moving function takes a robot ID and a coordinate. For now, coordinates are just pairs of numbers, since initially we focus on two-dimensional moves—but both the verification and the programming frameworks have been built in such a way that an upgrade to three dimensions can be made without much rewriting.

Finally, programs consist of multiple threads. We opted for a multi-threaded DSL (using the multi-threading facilities of Python) because this makes it easier to program collaborative tasks. Usually there is one thread dedicated to each participating robot, but our framework does not impose that—indeed, any thread can issue commands to any robot.

A thread consists of an initialization section (ISec) and a compound command (Com). In the initialization section, the global variables (visible in all the threads hence usable for inter-thread communication) are initialized with expressions, and the robots are initialized with "robot info" that has a format specific to each type of robot (e.g., various types of robotic arms or mobile platforms). Any variable that is not initialized as global is assumed to be local. The compound command is the actual code of the thread, written in a while language defined on top of the atomic commands.

This DSL, which (as mentioned) was implemented in Python on top of ROS, has the runtime behavior that one might expect. Thus, unless the code contains synchronization logic (e.g., waiting for a global flag to become true), the threads' commands are executed concurrently, so any ordering between the commands in different threads is possible. When a robot is asked for some information (via an API call in an expression), we can assume it will answer in a certain amount of time, so not necessarily instantly—any collaborative task program should take this into account, for example, the implemented protocol should take into consideration that a response to getPosition may become outdated. Similarly, when a "move to" command is issued (as discussed in §2.1.1) the robot will put this command in its queue and will get to it when it finishes the other commands from the queue (which it has received before).

The language does not contain timed commands. Moreover, we have no information about the speed with which a robot performs its tasks; but can only learn of the current status if we ask for it. These design decisions are intentional, since we aim for *collision-freeness gurantees that do not rely on time.*

2.2 Verification and Analysis Infrastructure

We have chosen the theorem prover Isabelle as the primary host of our verification infrastructure, because of its versatility and expressiveness. To enable verification, we will connect our programming environment with Isabelle. The identification of a suitable API and DSL for programming robot behaviors in a hassle-free manner (discussed in §2.1) has been a major step towards achieving this connection. Indeed, by isolating a relatively small fragment of Python and defining a simple interface to ROS-managed robots has created a manageable formalization task.

We have already specified a formal semantics of our API and DSL in Isabelle. To connect the Python/ROS DSL implementation with the Isabelle counterpart, we are implementing a translation (a code generator) between the two. This way, for example, the robot programmer can use the DSL to program the desired robot behavior, then the verification expert can employ the Python-to-Isabelle translation to obtain a copy of the program in Isabelle, where collision-freeness can be verified. This translation will be part of the trusted code base, and we will validate its soundness via testing.

From a formal perspective, collision freeness is a safety property ("something bad never happens") and has been defined as such in Isabelle. The formal semantics represents, in addition to the usual state containing values for the global and local variables, a "robot store" that keeps the status of each robot involved in the program—indicating whether the robot is currently stalling or moving between two waypoints, and the queue of tasks for which the robot has already received commands. In this context, collision freeness is formulated as follows: It is never the case that two robots are at the same time engaged in trajectories (i.e., pairs of waypoints) that can collide. The "can collide" predicate, which takes two trajectories (i.e., two pairs of positions) and returns true or false, is currently left generic in Isabelle. It will be instantiated to suitable concrete predicates based on domain-specific knowledge (obtained with the help of robot experts) about the type of the robot and its spatial and moving characteristics, and the known obstacles at the site. For example, if we have a room with only one gate and two trajectories whose start locations are outside the room and whose target locations are inside the room, then the “can collide” predicate will return true. Note that the “can collide” predicate will also able to accommodate properties such as ”physical” starvation, when two robots both need to enter some space but neither are ever able to.

3 (Very Rough Summary of) Related Work

The general area of robotic system safety assurance and verification is a large and rapidly growing area.[1] Important subareas include the verification of autonomous robotic systems [1, 4] and of industrial collaborative robots [2] using methods such as theorem proving, model checking and safety controller synthesis and monitoring. The employed formal models include timed automata [3] and process algebras [5]. Our work will mostly apply to robotic systems that are involved in pre-determined and largely pre-scripted collaborative tasks.

Acknowledgment. We thank the three anonymous reviewers for their valuable comments and suggestions, which led to the improvement of the presentation.

References

1. Clare Dixon (2020): *Verifying Autonomous Robots: Challenges and Reflections (Invited Talk).* In Emilio Muñoz-Velasco, Ana Ozaki & Martin Theobald, editors: *TIME 2020*, *LIPIcs* 178, pp. 1:1–1:4, doi:10.4230/LIPIcs.TIME.2020.1.
2. James A. Douthwaite, Benjamin Lesage, Mario Gleirscher, Radu Calinescu, Jonathan M. Aitken, Rob Alexander & James Law (2021): *A Modular Digital Twinning Framework for Safety Assurance of Collaborative Robotics.* *Frontiers Robotics AI* 8, p. 758099, doi:10.3389/frobt.2021.758099.
3. Raju Halder, José Proença, Nuno Macedo & André Santos (2017): *Formal Verification of ROS-Based Robotic Applications Using Timed-Automata.* In: *FormaliSE*, pp. 44–50, doi:10.1109/FormaliSE.2017.9.
4. Matt Luckcuck, M. Farrell, L.A. Dennis, C. Dixon & M. Fisher (2019): *Formal specification and verification of autonomous robotic systems: A survey.* *ACM Computing Surveys* 52(5), doi:10.1145/3342355.
5. Matthew O’Brien, Ronald C. Arkin, Dagan Harrington, Damian Lyons & Shu Jiang (2014): *Automatic Verification of Autonomous Robot Missions.* In: *Simulation, Modeling, and Programming for Autonomous Robots*, Cham, pp. 462–473.

[1] See https://www.andreipopescu.uk/litrev.pdf for a detailed literature review, including the connection with major verification projects such as those pursued at the RoboStar* center (https://robostar.cs.york.ac.uk).