Blending Kinematic and Software Models for Tighter Reachability Analysis

Carl Hildebrandt
The University of Virginia
ch6wd@virginia.edu

Sebastian Elbaum
The University of Virginia
selbaum@virginia.edu

Nicola Bezzo
The University of Virginia
nb6be@virginia.edu

ABSTRACT

Reachable sets are critical for path planning and navigation of mobile autonomous systems. Traditionally, these sets are computed using system models instantiated with their physical bounds. This exclusive focus on the physical bounds belies the fact that these systems are increasingly driven by sophisticated software components that can also bound the variables in the system models. This work explores the degree to which bounds manifested in the software can affect the computation of reachable sets, introduces an analysis approach to discover such bounds in code, and illustrates the potential of that approach on two systems. The preliminary results reveal that taking into consideration software bounds can reduce traditionally computed reachable sets by up to 91%.

CCS CONCEPTS

• **Computer systems organization** → **Robotics**; • **Software and its engineering** → **Software performance**; **Automated static analysis**.

KEYWORDS

Mobile Robots, Reachable Sets, Software Constraints

ACM Reference Format:
Carl Hildebrandt, Sebastian Elbaum, and Nicola Bezzo. 2020. Blending Kinematic and Software Models for Tighter Reachability Analysis. In *New Ideas and Emerging Results (ICSE-NIER'20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3377816.3381730

1 INTRODUCTION

As mobile autonomous systems increase in popularity, it becomes more critical to accurately predict the future states that may be covered by the system during its operation. Such predictions can assist a system in determining actions that lead to favorable states while avoiding detrimental ones[16]. Reachability analysis is one common approach to provide such predictions. Given a starting state and a time horizon, this analysis iteratively applies the set of admissible control sequences on a system model to compute the set of reachable states. Today, we find that the computation of reachable sets is at the center of many challenging tasks for mobile autonomous systems such as obstacle avoidance [13], aircraft collision avoidance [10], and system monitoring to maintain safety and liveness[15].

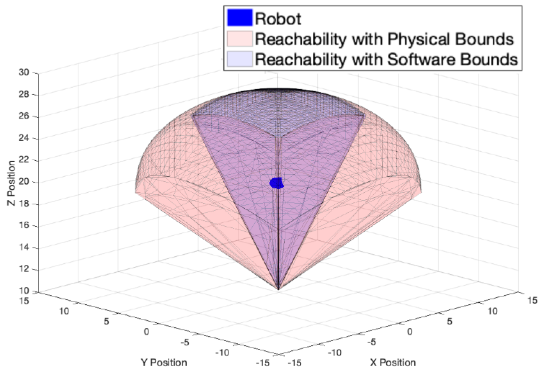


Figure 1: Reachable volume when parameterized with physical bounds (outer-lighter shade) and with the addition of software bounds (inner-darker shade), with a single time step $t = 1s$. Just considering the physical bounds results in an over-approximation of the robots reachable set.

There are many methods to compute reachable sets, ranging from numerical approximations to purely analytical methods . The choice of method depends on many factors, including what constitutes a state, what is the language of control actions, how long is the time horizon, whether under- or over-approximations are desirable, and the importance attributed to the efficiency of computation.[1]

Independent of the selected method, the calculation of the reachable sets is parameterized exclusively with the system's physical attributes. For example, for the differential drive robot we study later, the system's kinematic model[2] is instantiated with the robot's maximum physical velocity and turning rate. Similarly, when computing the reachable set for a quadrotor, which we study later, the physical attributes are used to bound pitch and roll.

This exclusive focus on the physical attributes of such systems belies the fact that these systems are increasingly driven by sophisticated software components that juxtapose another set of constraints on the system's potential state and behavior. *The insight of this work is that the precision of a reachable set could be dramatically higher by considering the constraints imposed by software.* For example, Figure 1 shows the reachable volumes of a quadrotor we later study, starting at a hovering state depicted with the central dark dot. The reachable set is computed using $t = 1s$, the volume in the outer-lighter shade is computed using physical bounds, and the inner-darker shade adds the bounds found in software. The

[1]For more details on reachable sets we refer the reader to the following papers [1, 6].
[2]A kinematic model is a mathematical model describing the motion of an object without considering the forces that lead to that motion.

dramatic reductions are achieved by instantiating the quadrotor kinematic model with just two bounds extracted from two software components (roll and pitch). These reductions result in more precise reachable sets which can ameliorate the safety and collision avoidance concerns of the mobile autonomous systems.

Building on this insight about the potential impact of imposing such software bounds on the reachable set, we propose a software analysis approach to refine the computed reachable sets. Given a target program variable that is known to influence the kinematic model (e.g., velocity, turning rate), we prototyped the approach using a depth-bounded inter-procedural interval analysis [8] to find the software bounds of possible values for that variable. Then, we combine the software and physical bounds to more accurately compute the reachable set of a mobile autonomous system and thus more accurately predict its possible future states.

We have performed a preliminary assessment of this approach on two systems. Our study revealed that many of the key variables in the kinematic models such as velocity, pitch, and roll are bound in the code, and that application of those bounds with short time horizons reduce the reachable sets by up to 91%.

2 PROBLEM

Reachability analysis is a well-known approach for predicting what future states a system may be in. Consider an autonomous system with general nonlinear dynamics $\boldsymbol{q}(t_{i+1}) = f(\boldsymbol{q}(t_i), \boldsymbol{u}(t_i))$, in which $\boldsymbol{q}(t_i)$ is the state of the system and $\boldsymbol{u}(t_i)$ the input at time t_i. Equation 1 describes a reachable set R computed from an initial time t_0 to t_e, where R is calculated based on the initial state, plant model f, and all valid inputs $\boldsymbol{U}$, with $\oplus$ representing the geometric sum.

$$R_i = \boldsymbol{q}(t_i) \oplus \int_{t_0}^{t_e} f(\boldsymbol{q}(t_i), \boldsymbol{U})dt \tag{1}$$

Our goal is to design a technique that can identify a set of software constraints C that can be applied to $\boldsymbol{U}$ to generate R'.

$$R_i' = \boldsymbol{q}(t_i) \oplus \int_{t_0}^{t_e} f(\boldsymbol{q}(t_i), C(\boldsymbol{U}))dt \tag{2}$$

The contribution of our technique is that we compute C such that $R_i' \leq R_i$ and C can be applied regardless of what approach was used to calculate R.

3 APPROACH

Figure 2 provides a conceptual overview of our approach compared with traditional reachability analysis. Traditionally, a kinematic model of a system is instantiated with physical bounds to generate a reachable set. In this work, we propose to uncover the software bounds that may affect the variables of the kinematic models.

Our approach takes the program under analysis p, one or more target variables v, v's coordinates (f$_n$; f$_l$) which represent filename and line number in the file, and depth d as input. The physical behavior of the robot is controlled by target variable v and is manifested in the kinematic model. For instance, in the case of the Husky robot [3] we later study, the variable *cmd_vel* is published to the robot and is the target program variable v that controls its physical velocity. Engineers can identify v in numerous ways, such as checking the standard control libraries output, inspecting the interfaces where messages are sent for actuation, or searching the documentation.

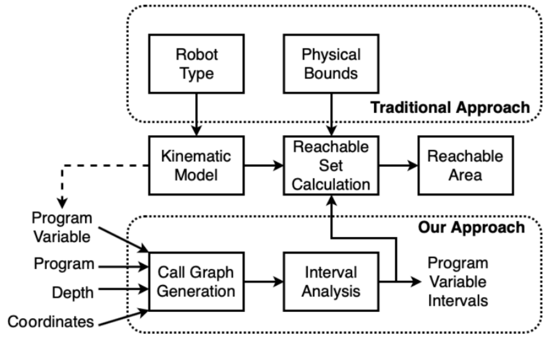


Figure 2: Conceptual overview.

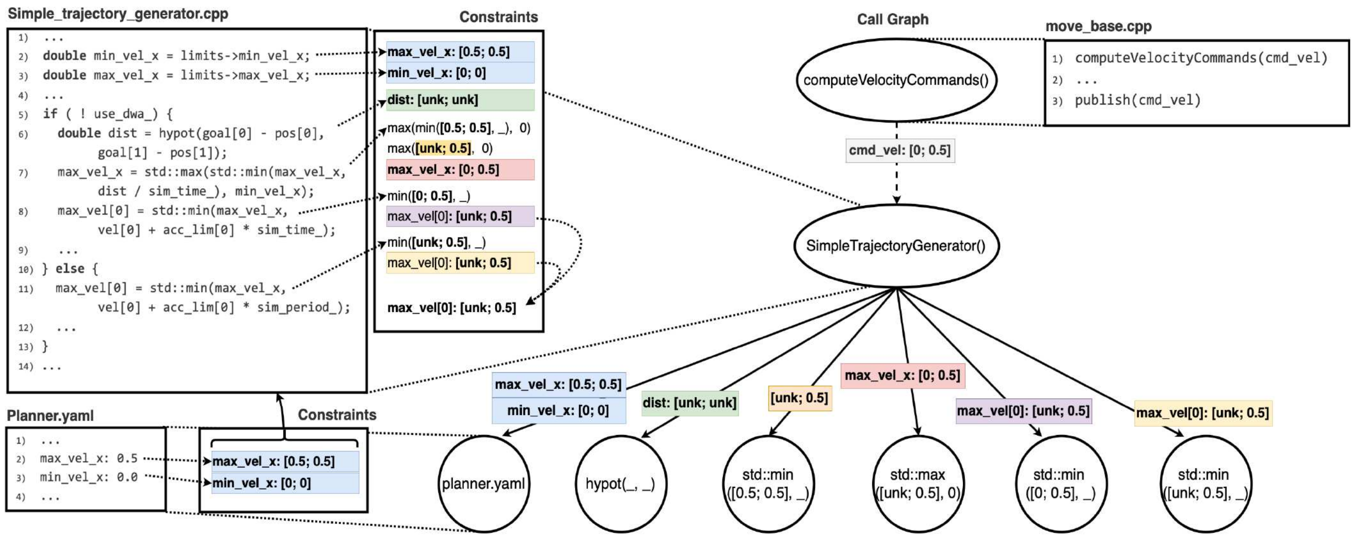
Algorithm 1: find_bounds

```
   Input: v, x, cur_depth
 1 bounds = [NaN, NaN];
 2 x.visited = True;
 3 if cur_depth < d then
 4     cur_depth ++ ;
 5     for node in x.callers and x.callees do
 6         if node.visited == false then
 7             b = find_bounds(v, node, cur_depth);
 8             bounds = bounds ∪ b
 9         end
10     end
11 end
12 final_bounds = interval_analysis(x, bounds);
13 return final_bounds(v) ;
```

The goal of our approach is to calculate conservative software bounds for all v using interval analysis. Interval analysis computes a lower and an upper bound for a given program variable[11]. Since applying interval analysis to entire programs is not always feasible nor scalable, our approach starts by computing a call graph for the robot's source code [2]. Using the target variable coordinates (f$_n$; f$_l$) allows the identification of the node x in the call graph, which contains the final assignment or publishing of v.

Algorithm 1 shows a pseudo-implementation of the approach. After computing the call graph and identifying x, the approach calls *find_bounds*. *find_bounds* recursively iterates d nodes away from x. Once the recursive search reaches depth d, it performs interval analysis on the node, taking into account any currently known bounds. As the call stack is resolved, the calculated bounds b are propagated back towards x by taking the union of the current bounds and calculated bounds. The union results in a conservative over-approximation of the interval by using the smallest lower bound and largest upper bound. Finally, the interval analysis is applied to x, taking into account the bounds propagated back from all callers and callees d calls away from x, resulting in intervals for all program variables in x including v.

For example, Figure 3 shows code from the ROS navigation stack used by the Husky robot. The Husky robot publishes a velocity command *cmd_vel* that controls the robots physical velocity. We start by computing the call graph, starting with target variable *cmd_vel* in line 3 of move_base.cpp. *cmd_vel* depends on function *computeVelocityCommands*, which becomes one branch of our call graph. Recursively traversing the call graph, we reach the base case,


Simple_trajectory_generator.cpp

```
1)  ...
2)  double min_vel_x = limits->min_vel_x;
3)  double max_vel_x = limits->max_vel_x;
4)  ...
5)  if ( ! use_dwa_) {
6)    double dist = hypot(goal[0] - pos[0],
          goal[1] - pos[1]);
7)    max_vel_x = std::max(std::min(max_vel_x,
          dist / sim_time_), min_vel_x);
8)    max_vel[0] = std::min(max_vel_x,
          vel[0] + acc_lim[0] * sim_time_);
9)    ...
10) } else {
11)   max_vel[0] = std::min(max_vel_x,
          vel[0] + acc_lim[0] * sim_period_);
12)   ...
13) }
14) ...
```

Planner.yaml

```
1)  ...
2)  max_vel_x: 0.5
3)  min_vel_x: 0.0
4)  ...
```

move_base.cpp

```
1)  computeVelocityCommands(cmd_vel)
2)  ...
3)  publish(cmd_vel)
```

Figure 3: Abbreviated code and call graph from a Husky showing the computation of a target variable (*cmd_vel*) intervals.

Figure 4: Erle-Copter and Husky [5][9]

which are calls to configuration files and standard library functions. For brevity, we show the analysis of leaf nodes in the call graph (base cases) and their parent node *simpleTrajectoryGenerator*.

The first base case which is reached is the configuration file planner.yaml, which assigns default values to *max_vel_x* and *min_vel_x*. Intervals for these variables have lower and upper bounds equivalent to the assigned constant. These intervals are returned to the previous function in the stack simple_trajectory_generator.cpp. Line 2 and 3 of simple_trajectory_generator.cpp are assignment statements, thus the intervals from planner.yaml are transferred with no modification. Line 7, makes calls to the *std::max* and *std::min* functions which are also base cases. Interval analysis of a *std::min* function with parameters 0.5 and an unknown value results in an interval with the upper bound set to 0.5 and the lower bound set to unknown. Similarly, interval analysis of *std::max* between 0.5 and 0 results in the upper bound staying at 0.5.

Interval analysis is not path sensitive and computes an interval for lines 8 and 11, similar to that of line 7. The union of the results from line 8 and 11 are used. In this case, they both have the same intervals and thus remain unchanged at $[unk, 0.5]$. This results in a $max\_vel[0]$ with an upper bound of 0.5. The lower bound for *min_vel* is computed using our approach and found to be 0. As the call stack is resolved, the intervals are transferred back to *cmd_vel*, which has an interval [0; 0.5]. Clearly, using this technique will result in tighter reachable sets when the robots software bounds limit the robots physical capabilities. This abbreviated example belies some of the complexity of traversing call chains and computing the intervals for target program variables. This approach does not account for potential unsoundness of the approach that could be addressed with more precise analysis.

4 EXPLORATORY STUDY

We conducted a preliminary study to investigate: 1) What bounds can be found in software that may be relevant to the systems kinematic models? 2) How different are software versus physical bounds? 3) What is the impact of those software bounds on the computed reachability sets?

4.1 Systems Overview

We use two robots for illustration. The first system is Husky from Clearpath Robotics [3] (right in Figure 4). Our analysis targets Husky's autonomous planning and exploration with simultaneous localization and mapping (SLAM) based on the frontier exploration software [14]. More specifically, our approach targets the move_base node in the navigation stack, which controls the physical attribute velocity. The second system was the Erle-Copter quadrotor [7] (left in Figure 4). It is open-source and comes with support for various controls and applications. The particular application selected allowed the drone to track and subsequently follow a ground marker by calculating pitch and roll values.

Both robots were deployed in an open world with no obstacles. We computed traditional reachability analysis based on the kinematic models for each robot. The robots initial position was $(0, 0)$ for the Husky and $(0, 0, 20)$ for the Erle-Copter. Kinematic inputs v were generated by creating 30 linearly spaced samples between each of the input bounds. The robots next state was then calculated using all permutations of the sampled input.

The kinematic models of the robots define the input target variables. Husky's inputs are velocity and turn rate. The physical bounds of the velocity are $[-1m/s,\ 1m/s]$ [4]. For Erle-Copter the maximum pitch and roll are set by the default maximum tilt parameter (MNT_ANGMAX_TIL) of the controllers firmware. The thrust value was calculated using the datasheet of the specific motors [12]. The physical bounds for both robots are in Table 1.

Table 1: A comparison of the physical and software bounds.

Robot Type	Physical Bounds	Software Bounds
Husky (Differential Drive)	Max Velocity: 1 m/s Min Velocity: -1 m/s Turn Rate: 2 rad/s	Max Velocity : 0.5 m/s Min Velocity: 0 m/s Turn Rate: 0.63 rad/s
Erle Quadrotor	Thrust: 45 N Max Pitch: 45 $degrees$ Max Roll: 45 $degrees$	Thrust: ? N Max Pitch: 19 $degrees$ Max Roll: 19 $degrees$

Table 2: Reachable set for physical and software bounds.

Robot type	Physically Bound Reachability	Software Bound Reachability	Reduction
Differential Drive ($t = 3s$)	$20.24m^2$	Max Velocity: $17.10m^2$	16%
		Min Velocity: $15.10m^2$	25%
		Velocity: $3.77m^2$	81%
		Max Turn Rate: $17.06m^2$	16%
		All Constraints: $1.85m^2$	91%
Quadrotor ($t = 3s$)	$716930m^3$	Max Pitch: $343428m^3$	52%
		Max Roll: $343428m^3$	52%
		All Constraints: $163563m^3$	77%

4.2 Preliminary Results

Our approach *found software bounds for 5 out of 6 target program variables*, the exception being Erle-Copter's Thrust, which was available but just not in the scope of files we analyzed. For Husky, the analysis traversed a call chain of 15 calls in order to capture all the software constraints, while for Erle-copter, the analysis only required the exploration of 2 function calls. In Table 1 we observe that there is a clear reduction from the physical to the software bounds, with software bounds being far tighter. An interesting finding illustrating the extent of those tighter bounds was found in the Husky robot, which is physically capable of going backward as fast as it can go forward, but our analysis revealed that the software would never publish negative velocities.

Given the identified physical and software bounds, we run two sets of simulations for 3-time steps ($\Delta t = 1s$) for both robots using the physical bounds and the software bounds. In both cases, the reachable set was calculated. Table 2 shows how the software constraint on each target program variable reduces the reachable set. For the differential drive robot, bounding velocity reduces the reachable area by 81%; bounding the turning rate reduces the reachable area by 16%; *applying all software constraints reduces the reachable area by* 91%. We find similar results for the Erle-Copter. Applying software constraints to either the pitch or roll reduced the reachable area by 52%. *Applying all software bounds results in a 77% reduction of the reachable volume.*

We now illustrate the impact of such reductions over time. The Husky physically bound reachable set is shown as dotted lines in Figure 5. The software bound reachable set, shown using solid lines, is contained within the reachable set computed with the physical bounds. Furthermore, the physical bounds reachable set grows much faster than the set computed with the software bounds. *For example, the Husky's software bound reachability at $t = 3s$ is a subset of the physically bounded reachable set at $t = 2s$.*

5 CONCLUSION

We have explored a different approach to compute reachable sets by leveraging the fact that mobile autonomous systems are increasingly affected by software bounds. Our preliminary findings indicate that the approach has significant potential to generate tighter

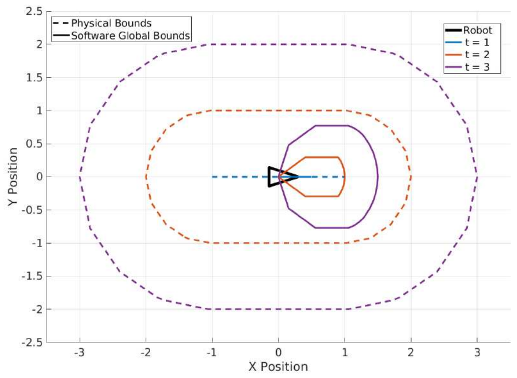


Figure 5: Husky reachable set using physical bounds (dotted lines) versus software bounds (solid lines) for $t = 3s$.

reachable sets, which can lead to safer, more effective planning and navigation. Future work will address the sources of imprecision in the approach by incorporated more sophisticated analyses. We also plan to determine the impact of such bounds through a more extensive assessment of our approach.

ACKNOWLEDGMENTS

This work is supported in part by the National Science Foundation Award #CCF-1718040. We thank the developers of Husky and Erle-Copter for making their systems and software available.

REFERENCES

[1] Somil Bansal, Mo Chen, Sylvia Herbert, and Claire J Tomlin. 2017. Hamilton-Jacobi reachability: A brief overview and recent advances. In *2017 IEEE 56th Annual Conference on Decision and Control (CDC)*. IEEE, 2242–2253.
[2] David Callahan, Alan Carle, Mary W. Hall, and Ken Kennedy. 1990. Constructing the procedure call multigraph. *IEEE Transactions on Software Engineering* 16, 4 (1990), 483–487.
[3] Clearpath Robotics. 2019. Husky - Unmanned Ground Vehicle. https://www.clearpathrobotics.com/husky-unmanned-ground-vehicle-robot/. [Online; accessed 27-February-2019].
[4] Clearpath Robotics. 2019. Husky Specifications. https://storage.pardot.com/92812/4030/Husky_DataSheet_2016.pdf. [Online; accessed 29-February-2019].
[5] Dabit Industries. 2019. Erle-Copter drone kit. https://dabit.industries/products/erle-copter-drone-kit. [Online; accessed 29-February-2019].
[6] Jerry Ding, Jeremy H Gillula, Haomiao Huang, Michael P Vitus, Wei Zhang, and Claire J Tomlin. 2011. Hybrid systems in robotics. *IEEE Robotics & Automation Magazine* 18, 3 (2011), 33–43.
[7] Erle Robotics. 2019. Erle - Copter. http://docs.erlerobotics.com/erle_robots/erle_copter. [Online; accessed 27-February-2019].
[8] William H. Harrison. 1977. Compiler analysis of the value ranges for variables. *IEEE Transactions on software engineering* 3 (1977), 243–250.
[9] IEEE. 2019. Robots - Husky. https://robots.ieee.org/robots/husky/. [Online; accessed 29-February-2019].
[10] Y. Lin and S. Saripalli. 2017. Sampling-Based Path Planning for UAV Collision Avoidance. *IEEE Transactions on Intelligent Transportation Systems* PP, 99 (2017), 1–14. https://doi.org/10.1109/TITS.2017.2673778
[11] Anders Møller and Michael I Schwartzbach. 2012. Static program analysis. *Notes. Feb* (2012).
[12] T-Motor. 2019. MN2212 KV920-V2.0 Specification. http://store-en.tmotor.com/goods.php?id=389. [Online; accessed 29-February-2019].
[13] Abraham P. Vinod, Baisravan Homchaudhuri, and Meeko M. K. Oishi. 2016. Forward stochastic reachability analysis for uncontrolled linear systems using Fourier Transforms. *CoRR* abs/1610.04550 (2016). http://arxiv.org/abs/1610.04550
[14] Brian Yamauchi. 1997. A frontier-based approach for autonomous exploration. In *cira*. IEEE, 146.
[15] E. Yel, T. X. Lin, and N. Bezzo. 2017. Reachability-based self-triggered scheduling and replanning of UAV operations. In *NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*. 221–228.
[16] Esen Yel, Tony X Lin, and Nicola Bezzo. 2018. Self-triggered Adaptive Planning and Scheduling of UAV Operations. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 7518–7524.