Submitted to *Mathematical and Computer Modelling of Dynamical Systems*
Vol. 00, No. 00, Month 20XX, 1–21

Mathematical and Computer Modelling of Dynamical Systems

Supplemental online material for the paper "Automated generation of hybrid automata for multi-rigid-body mechanical systems and its application to the falsification of safety properties"

E.M. Navarro-López[a*] and M.D. O'Toole[b]

[a]*School of Computer Science, The University of Manchester, Oxford Road, Kilburn Building, Manchester M13 9PL, UK;* [b]*School of Electrical and Electronic Engineering, The University of Manchester, Sackville Street Building, Manchester M13 9PL, UK*

(*August 2017*)

In this document, we provide details of the tool DyverseBMC which uses the modelling framework proposed in our paper for the falsification of safety properties of multi-rigid-body systems with multiple contacts, impacts and discontinuous friction. We also make a summary of the main limitations of the implemented methodology and explore some of these limitations for a multiple contact problem.

Keywords: hybrid systems; hybrid automata models; design automation; computational methods; computer simulation

AMS Subject Classification: 93C30; 68Q60; 68U20; 93B40; 70E55

Contents

---

*Corresponding author. Email: eva.navarro@manchester.ac.uk. This work was supported by the Engineering and Physical Sciences Research Council (EPSRC) of the UK under Grant EP/I001689/1 ('DYVERSE: A New Kind of Control for Hybrid Systems'); and the Research Councils UK under Grant EP/E50048/1.

1. Dyverse bounded model checker and the MRB hybrid automaton

In this document, we describe DyverseBMC: a procedure for falsifying a safety property over finite time-intervals for a rigid body system against our multi-rigid-body (MRB) hybrid automaton abstraction. DyverseBMC has to be understood as a wider modelling, simulation and bounded-model-checking framework which includes the specification of the computational semantics for a multi-rigid-body system. This specification is given by the MRB hybrid automaton formalism and is explained in the main paper. The automatic generation of the MRB hybrid automaton and its simulation are implemented through the tool DyverseRBT. The main elements of our framework were given in Figure 1 of the main paper.

2. Overview of the bounded model checking approach

The falsification method presented as an application of our modelling framework is based on the concept of bounded model checking (BMC), and the observation that a designer or engineer will manually check a system's property by repetitive simulations, using their own reasoning processes to guide the parameters of the simulation towards a desired, or in this case, unsafe result. A mix of new and well-established methods for the simulation of non-smooth mechanical systems are used to discretise the system. The discretised system is then posed as a formula consisting of a Boolean combination of interval-based arithmetic constraints. Automated reasoning tools such as interval constraint solvers [1, 2] can be used to efficiently find solutions to the formula which violates our specified safety property. If no unsafe solutions are found, we extend the length of the discretisation in time by adding new time-steps. The procedure is thus akin to the BMC approach, which has been used with considerable success as a pragmatic approach to the formal verification of software and embedded systems. In brief, we translate the mechanical system dynamical evolution into SMT (Satisfiability Modulo Theory) formulas to apply an SMT-type solver which has been especially tailored for the type of systems we are dealing with. We point out that under this framework, the continuous dynamics of the system (ODEs) are interpreted as arithmetic constraints over real-valued variables, which are nonlinear in our case. Additional constraints are the guards and the domains of each discrete location. Due to the geometry of the bodies considered (spheres), the guards are considered to be convex and the domains are concave.

To put it in a nutshell, we present a method of using the multi-rigid-body (MRB) hybrid automaton to generate a succession of mixed Boolean/arithmetic constraint satisfaction problems which can be used with a lazy-SMT solver to find unsafe trajectories over progressively lengthening time-scales. The custom translation to the MRB hybrid automata is a necessary part of this process. In fact, we find natural synergies between the different elements of the automaton and posing constraint satisfaction problems associated with trajectories of the state-space. For example, the guard conditions of the hybrid automaton are already examples of mixed Boolean/arithmetic constraints which define whether the trajectory has left the location.

To solve the SAT (Boolean satisfiability) problem, we use optimisation techniques. Inspired by [3], we perform a depth-first search through the space of possible variable assignments. The optimisation problem we solve is the sum-of-slacks feasibility problem [4]. The implementation is done using the NAG Toolbox for MATLAB®

[5] for the optimisation, and the SAT-solver of MATLAB®.

Our falsification approach is based on a methodology akin to bounded model checking briefly described in the Introduction and Section 4 of the main paper.

A logical formula is constructed consisting of Boolean variables and constraints on real variables. This formula is satisfied only by a certain set of trajectories across the continuous state-space over finite time-intervals.

A lazy SMT solver [2, 6] is used to find a solution to this formula which falsifies the safety property. If no falsifying solution exists, then we can conclude that there are no trajectories amongst that set which are unsafe over the time interval. We then move to a different set of trajectories, and repeat the search for unsafe trajectories.

The logical formulas which we will construct are in conjunctive normal form (CNF) and consist of mixed Boolean variables and constraints on real variables, that is,

$$\varphi := \bigwedge_{i \in I} \left(\bigvee_{j \in J_i} p_{i,j}(a, b, ..., c(x_1, x_2, ...) \circ 0, ...) \right),$$

where $p_{i,j}(a, b, ..., c(x_1, x_2, ...) \circ 0, ...)$ is either a Boolean variable $a, b, ...$, or a constraint $c(x_1, x_2, ...) \circ 0$ on the real variables $x_1, x_2, ...$, where $\circ := \{=, \leq\}$. For example, we might have,

$$\varphi := (a \vee b) \wedge (x_1 = 0 \vee x_2 \leq 0)$$

where $x_1, x_2 \in \mathbb{R}$ and a, b are Boolean variables. A solution to a formula φ is any choice of variables (real and Boolean) such that φ evaluates to true.

The relational operator $\circ$ of the constraint is restricted in this manner $\circ := \{=, \leq\}$. We deliberately do not allow the operator $<$. This restriction is imposed by the standard numerical optimisation tools which we use to assist in finding solutions to the formula. This will be described in more detail in Section 6.

To verify a safety property is to prove that it holds for a given system. We have already defined what we mean by a safety property and its restrictions in the introduction. To reiterate, we have the following components,

- ϕ: A CNF formula as defined above.
- H_{MRB}: A multi-rigid body mechanical system with continuous states $x(t)$.
- T: A finite time interval.
- $I(x(0))$: A set of initial conditions of the state variables.
- P: A range of possible values for the physical parameters in the system (mass, dimensions, coefficients of friction or restitution, etc).

For the safety property to hold, ϕ must be satisfied for all $x(t)$, such that $t \in T$, $x(0) \in I$, and for all physical parameters across P. Our procedure will find solutions which falsify the safety property.

The overall procedure is shown in Algorithm 1. The algorithm uses a depth-first-search method to build possible discrete evolutions, which are then checked for safety. More specifically, the discrete evolution is checked for the existence of a continuous trajectory that starts in the initial conditions, visits each discrete location in order, and at some point, violates the safety property.

For instance, the discrete evolution $s_1 \rightarrow s_2 \rightarrow s_3$ is unsafe, if there exists a continuous trajectory which starts from a set of initial conditions in the domain of

the discrete location s_1, then enters the domain of s_2, followed by the domain of s_3, and at some stage enters a region of the state space specified as unsafe.

If the discrete evolution is determined to be safe, then a new location is added to the evolution following the depth-first-search method. This new and longer discrete evolution is then checked for safety, and so on, until an unsafe solution is found, or until some chosen maximum depth is reached.

The safety verification task is conducted using the function *ExploreDynamicalLocation*. The functions *CreateRoot* and *ExploreCompNode* are used to aid in constructing formulas for trajectories that traverse between discrete locations and domains. The purpose and details of these functions will be made clear in subsequent sections.

Algorithm 1 DepthFirstSearch (Falsification of a safety property)

Input: s, $root$
if s is a dynamical location **then**
  $(safe, roots) \leftarrow ExploreDynamicalLocation(root, f_s(x), Dom_s, \phi)$
else
  $ExploreCompNode(root, s)$
end if
for all $\varphi \in roots$ (for all formula in $roots$) **do**
  $\varphi_{s'}$ = root formula for location s'
  $safe \leftarrow DepthFirstSearch(s', \varphi_{s'})$
  if $safe = false$ **then**
    return
  end if
end for
Output: variable $safe$ (if $safe = false$, the system is unsafe)

3. Exploring the dynamical discrete locations

The purpose of the function *ExploreDynamicalLocation* is to construct formulas which are true for sets of trajectories that cross the domain of some dynamical discrete location. Thus, by finding intersection with the complement of the safety property – the unsafe region – we can verify whether the system is safe up until some moment in time. Algorithm 2 shows the procedure used by this function.

To explain this procedure, we must first consider the following preliminaries. We denote the dynamical location we are attempting to verify as s and E_s is the set of all edges leaving this location. We also define the following CNF formulas, which parallel the elements of the MRB hybrid automaton (defined in Section 3 of the main paper):

- $Dom_s(x^{(i)})$ is a CNF formula which is true if $x^{(i)}$ in the state space belongs to the domain of location s, that is, $x^{(i)} \in Dom(s)$.
- $\mathbf{G}_e(x^{(i)})$ is a CNF formula which is true if $x^{(i)}$ in the state space belongs to the guard set associated with edge e, that is, $x^{(i)} \in G(s, s')$ if transition e is $s \to s'$, $e = (s, s')$.
- $\phi(x^{(i)})$ is a CNF formula which is true if $x^{(i)}$ satisfies the safety property.
- $root$ is a CNF formula which is true for all trajectories that start from a set

of initial conditions and have a common discrete evolution, that is, they all traverse the same ordered set of edges, $s \to s' \to s'' \to ...$ etc.

Further, we introduce a function $ODE(f, x_0, \Delta T)$, representing the solution $x(\Delta T)$ of the initial value problem $\dot{x}(t) = f(x(t))$, $x(0) = x_0$. Let $x^{(i)}$ be the sequence $x^{(i+1)} = ODE(f, x^{(i)}, \Delta T)$, starting from a given initial value $x^{(1)}$.

We describe the falsification procedure using the example shown in Figure 1. The example assumes that only five trajectories (A, B, C, D and E) are possible and that they evolve from the set of initial conditions indicated in the figure. Clearly, this is not valid generally, as a dynamical system can have possibly infinite trajectories evolving from a bounded region. However, this simplification serves clarity, and this assumption is only for illustrative purposes.

The example contains two guard sets, which we designate $G_1 = G(s, s')$ and $G_2 = G(s, s'')$, and two accompanying guard formulas $\mathbf{G}_1$ and $\mathbf{G}_2$, such that any point belonging to a guard set is also a solution to the respective guard formula. The domain $Dom(s)$ is the region bounded by the guards on the top and right hand-side. The accompanying formula is Dom_s. Clearly, any state x' in this space is a solution to the formula $Dom_s(x')$.

We further introduce an unsafe region. This is determined from the complement of the safety property. A state x' inside the unsafe region is a solution of the formula $\neg\phi(x')$.

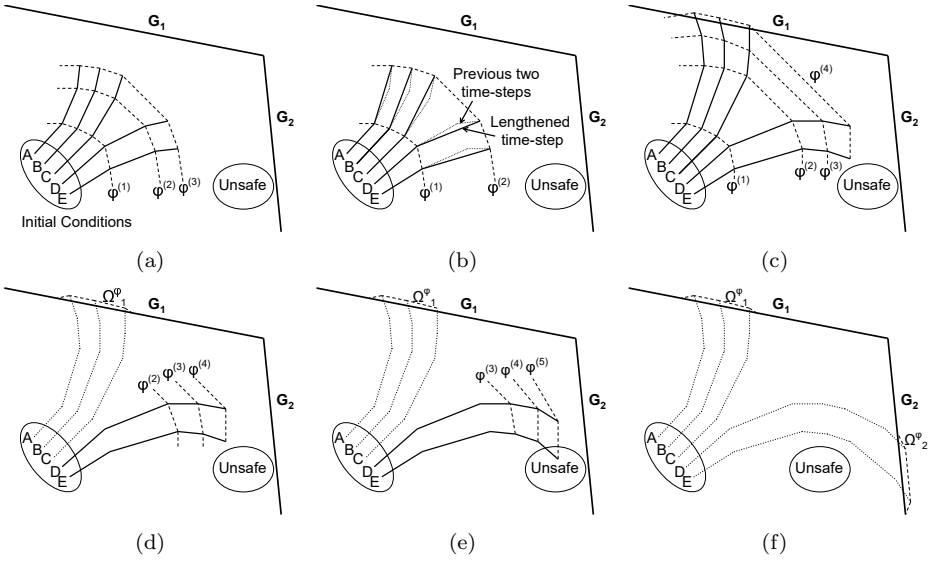


Figure 1. Exploring the dynamical discrete location s.

All the trajectories starting from the initial conditions up to the first and second dashed line in Figure 1(a) are solutions to the formulas $\varphi^{(1)}$ and $\varphi^{(2)}$, respectively. For our explanation, we shall assume that the first two steps have been verified safe, that is, all trajectories up to the second dashed line do not intersect the unsafe region. We are observing the falsification procedure mid-way through its evaluation of this location.

We add a new step to the length of our trajectories $x^{(3)}$, and create a new formula $\varphi^{(3)}$ by appending the equality constraint $x^{(3)} = ODE(f_s, x^{(2)}, dt)$ to the previous formula, where f_s is the vector field governing the dynamics at location s (see the hybrid automaton definition in the previous section) and dt is some small time-step.

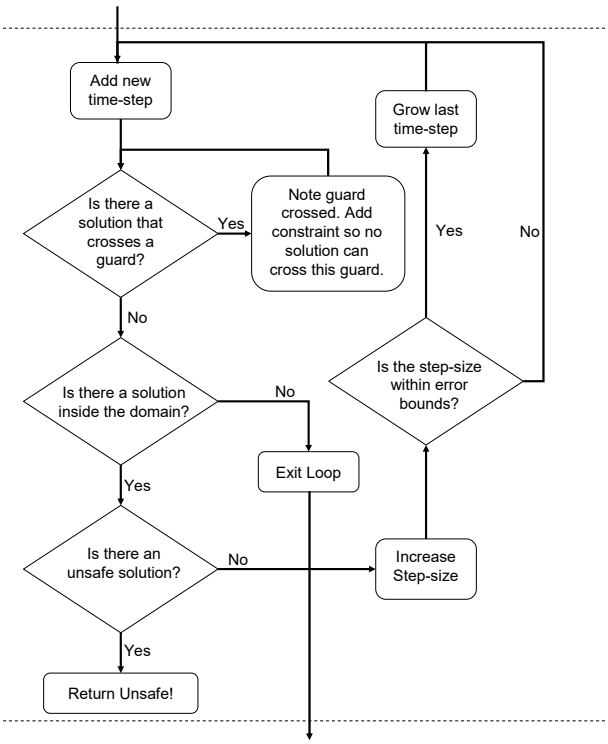


Figure 2. Flow diagram: how a new time-step affects the trajectories.

All trajectories starting from the initial conditons and going to the end points on the third dashed line in Figure (1(a)) are satisfiable solutions of $\varphi^{(3)}$. Our task is to analyse how this new time-step affects the trajectories (up to $x^{(3)}$) by applying a series of tests:

(1) *Do any of the possible trajectories enter a guard set?* In other words: is there a solution of $\varphi^{(3)}$ such that one or more of of the guard formulas are also true?
(2) *Are there any trajectories still evolving within the domain of s?* That is: is there a solution $\varphi^{(3)}$ such that $Dom_s(x^{(3)})$ is also true? If no solution is possible then we have 'exhausted the domain', and we can reason that all trajectories must have left the domain, and that no further exploration of this location is necessary. Figure 1(f) is an example of this, which we will discuss later on.
(3) *Are there any trajectories which violate the safety property by entering the unsafe region?* In other words: are there any solutions to $\varphi^{(3)}$ and where the safety property does not hold?

These tests are summed up in Figure 2. From Figure 1(a), it is clear that, for all trajectories up to $x^{(3)}$, none of the guard sets are entered, the domain is not exhausted, and the safety region is not violated. The new time-step has been uneventful.

The addition of each new time-step increases the number of clauses contained in our formula. Larger formulas increase the solving time. It is thus desirable to minimise the number of time-steps, and maximise the length in time of our trajectories without drastically impinging on the accuracy.

We implement a variable-step scheme, which grows the step-length whenever a new time-step proves uneventful. The new time-step is temporarily discarded, and the previous time-step has the step-size increased so that the trajectory has the same length in time for fewer time-steps. In Figure 1(b), for example, we see that each of the trajectories reach the same point as they did in Figure 1(a), but with only two time-steps rather than three.

A new formula $\varphi^{(2)}$ is created using the adjusted step-sizes, such that all trajectories in Figure 1(b) are solutions of $\varphi^{(2)}$. The integration errors for the possible solutions of $\varphi^{(2)}$ are examined. If there is no solution to $\varphi^{(2)}$ which results in an integration error above a given tolerance, then the new formula $\varphi^{(2)}$ and the new lengthened time-step is accepted. If a solution is found with an error above the tolerance, then the lengthened time-step is discarded and we revert back to having two separate time-steps as in Figure 1(a).

From Figure 1(b), we advance the trajectories by two time-steps to produce the result in Figure 1(c). The new time-step causes the trajectories A, B, and C to enter the guard set G_1. We take note of the following:

- The edge e that is associated with the guard set that has been entered, in this case e_1;
- the formula before the addition of the new time-step, in this case $\varphi^{(3)}$;
- an over-approximation of the time between the first trajectory and the last trajectory enters the guard set.

In the case of Figure 1(c), we know that all trajectories that are going to enter G_1 enter during the period of a single time-step. Thus, our over-approximation is the length of the time-step.

This information is recorded in the set Ω_1^{φ}. This set will be used in Section 4 to create formulas for successive locations in the discrete evolution of the system.

We check if any other guard region is entered by temporarily including the constraint $\neg\mathbf{G}_1(x^{(4)})$ so that any solution which enters $\mathbf{G}_1$ is excluded. The search for solutions is then repeated. In Figure 1(c), no other guard sets are entered during this time-step so the formula $\varphi^{(4)}$ with the additional guard constraint $\neg\mathbf{G}_1(x^{(4)})$ is unsatisfiable. The guard constraint is discarded and the process continues.

We are no longer interested in the trajectories that have entered the guard set as these have left the domain which we are exploring. To exclude these trajectories, an additional set of clauses is added to the formula $\varphi^{(4)}$,

$$\varphi^{(4)} := \varphi^{(4)} \ \wedge \ Dom_s(x^{(4)}),$$

such that the solutions of $\varphi^{(4)}$ become only those where the fourth point $x^{(4)}$ is inside the domain of s. The fourth points are those points along the trajectories that lie on the dashed line marked $\varphi^{(4)}$ in Figure 1(c). Consequently, the trajectories A, B, and C are excluded from $\varphi^{(4)}$. We are not interested in these trajectories for the present as they have exited the location and have not been found to be unsafe.

A new time-step is added, and a new formula $\varphi^{(5)}$ is created from $\varphi^{(4)}$ – recall $\varphi^{(4)}$ still includes the domain condition on $x^{(4)}$. The solutions of $\varphi^{(5)}$ are the lengthened trajectories D and E in Figure 1(e), but not A, B, and C as they are excluded by the domain condition.

The new step does not result in any of the remaining trajectories entering a guard set, and both trajectories exist in the domain of s. However, as shown in Figure 1(e), the distal point of trajectory E has entered the unsafe region. Thus, a solution

exists to the formula,

$$\varphi^{(5)} \ \wedge \ \neg\phi(x^{(5)}).$$

The conclusion is that the system is unsafe and the algorithm returns this result.

Figure 1(f) shows a different circumstance. In this case, the trajectories avoid the unsafe region, and eventually enter the guard set $\mathbf{G}_2$. There are no trajectories which continue to evolve in the domain of s. Thus, as discussed previously, we have exhausted the domain, and the exploration for *this location* is terminated. The evolution of the system so far is judged 'safe'.

4. Creating the 'root' formulas

Consider a transition evolution, from one location to another, of an execution of the MRB hybrid automaton as $s \to s' \to s''$. If we wish to verify a property while the system is in the last of these locations we must create a formula which has as its solutions, all trajectories that start from the initial conditions and follow the same path of transitions until reaching the last location. We call this formula the 'root' of location s''.

The root is used in the same ways as the initial condition set in Figure 1. That is, there is a region in the domain of s'' which consists entirely of solutions to the root formula, and from which all trajectories of interest evolve. By replacing the initial condition region in Figure 1 with 'root', the procedure is the same.

The method for constructing root formulas is described following the example in Figure 1(c). We found that the trajectories A, B, and C, were able to enter the guard set G_1 within one time-step. Recall that we recorded $\Omega_1^{\varphi} = \{1, \varphi^{(3)}, \Delta T\}$, where 1 is the edge label, $\varphi^{(3)}$ is the formula prior to the addition of the 'guard-entering' time-step, and ΔT is an over-approximation of the time between the first and last trajectory entering the guard set G_1. We call the over-approximation ΔT the 'crossing-period'.

Let us say that edge 1 is the transition between locations $s \to s'$. Algorithm 3 of 'CreateRoots' is used to create the root of location s'. We add a new time-step to our formula using the ODE function. However, in contrast to our previous method, we add a time-step with a symbolic variable to represent the size of the time-step and add constraints to allow this variable to take any value over the range $[0, \Delta T]$.

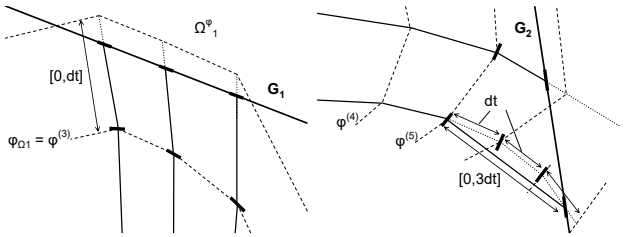
The new time-step creates a new point $x^{(4)}$ which we constrain to the boundary of G_1 using a modified version of $\mathbf{G}_1(x^{(4)})$ with inequalities replaced by equalities. The result is shown in Figure 3(a). The point $x^{(4)}$ sits on the boundary of G_1. The step between $x^{(3)}$ and $x^{(4)}$ has a variable step-size and can 'stretch' so that the small time-step will always enter the guard.

The resulting formula, designated $\varphi_{\Omega,1}$ in Algorithm 3, is the root of location s' for the transition $s \to s'$. A trajectory is a solution of $\varphi_{\Omega,1}$ if it starts from the initial conditions and enters the guard set G_1 without entering any other guard set or visiting any other locations.

An alternate scenario is shown in Figure 3(b). In this case, all trajectories that are going to enter the guard set G_2, do so over a period of three time-steps, rather than a single step as before. This procedure is the same, except that we allow the time variable in the new time-step to range over $[0, 3\Delta T]$, and point $x^{(6)}$ is constrained to the boundary of G_2.

Algorithm 2 *ExploreDynamicalLocation*

Inputs: $root$, s **(location)**
$dt_d \leftarrow$ Time-step increment size
$i \leftarrow$ maximum number of time-steps in $root$ (i.e. highest superscript of x)
$E_s \leftarrow$ index of edges leaving location s
$\varphi^{(i-2)} \leftarrow \varphi^{(i-1)} \leftarrow root$
$\Omega^{\varphi} \leftarrow \left\{..., \Omega_j^{\varphi} = \{\emptyset\}, ...\right\}\ \forall j \in E_s$
$t_{clk} \leftarrow 0,\ i \leftarrow i+1\ dt \leftarrow 0,\ safe \leftarrow true$
while the minimum length trajectory (in time) is less than the upper-bound of the interval T of the safety property **do**
  $\varphi^{(i)} \leftarrow \left(\varphi^{(i-1)} \wedge x^{(i+1)} = ODE(f_s, x^{(i)}, dt_d)\right)$
  • Is there a solution such that the new time step crosses any of the guards?
  $I \leftarrow E_s$
  $J \leftarrow \{\emptyset\},\ sat \leftarrow true$
  while sat **do**
    $sat \leftarrow Satisfy\left(\varphi^{(i)} \wedge \left(\bigvee_{\forall j \in I} G_j\right) \bigwedge_{\forall k \in J} \neg G_k\right)$
    if sat **then**
      (Denote e as the edge that is crossed)
      if $\Omega_e^{\varphi} = \{\emptyset\}$ **then**
        $\Omega_e^{\varphi} \leftarrow \left\{e, i, \varphi^{(i-1)}, t_{clk}\right\}$
      end if
      $\Omega_e^{\varphi} \leftarrow \Omega_e^{\varphi} \cup (t_{clk} + dt)$
      $I \leftarrow I \setminus e$
      $J \leftarrow J \cup e$
      $event \leftarrow true$
    end if
  end while
  • Is there a solution such that the new time step is inside the domain?
  if $Satisfy\left(\varphi^{(i)} \wedge Dom_s(x^{(i)})\right)$ **then**
    • Is there a solution such that the new time-step is unsafe?
    if $Satisfy\left(\varphi^{(i)} \wedge \neg\phi(x^{(i)})\right)$ **then**
      return $safe \leftarrow false$
    end if
  else
    • Exit the while loop (there are no more time-steps in this domain)
  end if
  if $\neg event$ **then**
    $dt \leftarrow dt + dt_d$
    $\varphi^{(i-1)} \leftarrow \varphi^{(i-2)} \wedge x^{(i)} = ODE(f_s, x^{(i-1)}, dt)$
  else
    $t_{clk} = t_{clk} + dt - dt_d,\ dt \leftarrow dt_d,\ i \leftarrow i+1,\ event \leftarrow false$
    $\varphi^{(i-2)} \leftarrow \varphi^{(i-1)}$
    $\varphi^{(i-1)} \leftarrow \varphi^{(i)} \wedge Dom_s(x^{(i)})$
  end if
end while
$roots \leftarrow CreateRoots(\Omega^{\varphi})$
Output: $safe\ roots$



(a) (b)

Figure 3. Creating root formulas.

Algorithm 3 *CreateRoots*

Inputs: Ω^{φ}

$\Omega^{\varphi} := \left\{..., \Omega^{\varphi}_j, ...\right\}$

$roots \leftarrow \{\emptyset\}$, $I \leftarrow \{\emptyset\}$

for all $\left\{j : \Omega^{\varphi}_j \in \Omega^{\varphi},\ \Omega^{\varphi}_j \neq \{\emptyset\}\right\}$ **do**

$\Omega^{\varphi}_j := \left\{\varphi_{\Omega_j}, i_{\Omega_j}, t^-_{clk,j}, t^+_{clk,j}\right\}$

$T \leftarrow 0$

$\Delta T \leftarrow t^+_{clk,j} - t^-_{clk,j}$

$\varphi_{\Omega_j} \leftarrow \left(\varphi_{\Omega_j} \ \wedge\ x^{(i_{\Omega_j}+1)} = ODE(f_s, x^{(i_{\Omega_j}+1)}, [0, \Delta T])\right)$

$\varphi_{\Omega_j} \leftarrow \left(\varphi_{\Omega_j} \ \wedge\ \text{boundary of } G_j(x^{(i)})\right)$

$i_{\Omega_j} \leftarrow i_{\Omega_j} + 1$

$roots \leftarrow roots \cup \varphi_{\Omega_j}$

$I \leftarrow I \cup i_{\Omega_j}$

end for

return $roots$, I

5. Exploring the computation nodes

We now consider the case of transitions from a dynamical location s to the entrance location of an impact or contact computation nodes, $s \rightarrow \mathcal{I}_{en}$ or $s \rightarrow \mathcal{C}_{en}$, and the return transition, from an exit location to a dynamical location, $\mathcal{I}_{ex} \rightarrow s'$ or $\mathcal{C}_{ex} \rightarrow s'$. The objective is to find a formula which is satisfied by any valid combination of contact forces given the possible trajectories which could trigger an impact or contact situation. This is relatively straightforward for the single contact case. However, for multiple contacts, the mutual interdependence of the contact forces between different contact sites makes the problem much more difficult. There is no general closed-form solution to computing forces in multiple contact cases. We must instead resort to iterative numerical procedures.

When simulating rigid-body contacts, the successive over-relaxation proximal point method – SORPROX, [7] – has been found to efficiently compute contact forces for rigid-bodies with multiple contacts. This method is briely reviewed in the appendix. We wish to follow this method, but with the iterative equations designed to compute contact forces for a single trajectory replaced by a formula satisfied by sets of trajectories. The formula equivalent for a single iteration of the SORPROX

method is,

$$\left(\Phi_{N,j} < 0 \ \vee \ f_{N,j}^{(i)} = \Phi_{N,j}\right) \ \wedge \ \left(f_{N,j}^{(i)} = 0 \ \vee \ \Phi_{N,j} \geq 0\right) \ \wedge$$
$$\wedge \ \left(\|\Phi_{T,j}\| \leq \mu_j \lambda_{N,j}^{(i)} \vee \lambda_{T,j}^{(i)} = -\mu_j \lambda_{N,j}^{(i)} \frac{\Phi_{T,j}}{\|\Phi_{T,j}\|}\right),$$

where the subscript j is the contact index, the superscript (i) is the current iteration, the terms $\Phi_{N,j}, \Phi_{T,j}$ are 'guesses' for the contact forces and are defined by equations (A1)-(A2) for computation node $\mathcal{I}$ (impact computation node), or equations (A3)-(A4) for computation node $\mathcal{C}$ (contact computation node). The terms $\lambda_{N,j}$, $\lambda_{T,j}$ are the normal and tangential contact forces, respectively.

The formula is constructed using Algorithm 4 of 'ExploreCompNode'. The algorithm appends the above clauses to φ for each contact j associated with the computation node. Next, the algorithm adds a set of convergent conditions. If these conditions are satisfied then the contact forces have converged to within a tolerance (tol) and no further iterations are necessary. First, a copy of φ is made and is called φ', so we can remove the convergent conditions if we require more iterations. Then, a new set of clauses are appended to φ' for each contact j. These clauses are equivalent to the complement of the convergence criteria in the SORPROX algorithm,

$$\left|f_{N,j}^{(i)} - \Phi_{N,j}\right| \geq tol \ \wedge \ f_{N,j}^{(i)} \neq 0 \ \wedge \ \left\|f_{T,j}^{(i)} - \Phi_{T,j}\right\| \geq$$
$$\geq tol \ \wedge \ \left(\|\Phi_{T,j}\| \leq \mu_j f_{N,j}^{(i)} \ \vee \ \left\|f_{T,j}^{(i)}\right\| < \mu_j f_{N,j}^{(i)}\right).$$

If there exists a solution to φ', i.e. it is satisfiable, then there exists a trajectory where the contact forces have not converged to a solution and the while-loop repeats to find the next iteration of the contact forces. Alternatively, if φ' is unsatisfiable, then we have sufficient iterations to guarantee all trajectories which entered the computation node have correct contact forces. Finally, when convergence is completed, the algorithm adds new clauses that assign the resets to the state variables, as defined in our hybrid automata specification, and test the new assignment for unsafe solutions and guard crossings, as previously in *ExploreDynamicLocation*.

6. Lazy SMT solver

In this section, we describe our implementation of a 'Lazy SMT solver' designed specifically to solve the mixed Boolean and real non-linear arithmetic formulas constructed in the previous sections. We use a basic method termed the *off-line approach* by [6], which is used by existing packages such as Absolver [2], among others. We include a number of additional heuristics and features on top of this basic method to increase efficiency and reduce solving times.

Algorithm 5 of 'Satisfy' shows the Lazy SMT method. Broadly, it works by first generating a Boolean abstraction of the formula to be solved. For example, consider the formula,

$$c_1(x) \leq 0 \wedge (c_2(x) \leq 0 \vee c_3(x) = 0) \wedge (c_1(x) \leq 0 \vee c_3(x) = 0).$$

Algorithm 4 *ExploreCompNode*

Inputs: $root$, $\mathcal{I}$ or $\mathcal{C}$ (computation node)
$\varphi \leftarrow root$
$sat \leftarrow true$
$i \leftarrow$ maximum number of time-steps in $root$ (i.e. highest superscript of x)
while sat **do**
  $i \leftarrow i+1$
  for $j = 1 \rightarrow$ Number of contacts **do**
    $\varphi \leftarrow \varphi \ \wedge \ \left(\Phi_{N,j} < 0 \ \vee \ f_{N,j}^{(i)} = \Phi_{N,j}\right)$
    $\varphi \leftarrow \varphi \ \wedge \ \left(f_{N,j}^{(i)} = 0 \ \vee \ \Phi_{N,j} \geq 0\right)$
    $\varphi \leftarrow \varphi \ \wedge \ \left(\|\Phi_{T,j}\| \leq \mu_j \lambda_{N,j}^{(i)} \vee \lambda_{T,j}^{(i)} = -\mu_j \lambda_{N,j}^{(i)} \frac{\Phi_{T,j}}{\|\Phi_{T,j}\|}\right)$
    $\varphi \leftarrow \varphi \ \wedge \ \left(\|\Phi_{T,j}\| > \mu_j \lambda_{N,j}^{(i)} \ \vee \ \lambda_{T,j}^{(i)} = \Phi_{T,j}\right)$
  end for
  • Does there exist a solution that does not converge?
  $\varphi' \leftarrow \varphi$
  for $j = 1 \rightarrow$ Number of contacts **do**
    $\varphi' \leftarrow \varphi' \ \wedge \ \left|f_{N,j}^{(i)} - \Phi_{N,j}\right| \geq tol \ \wedge \ f_{N,j}^{(i)} \neq 0$
    $\varphi' \leftarrow \varphi' \ \wedge \ \left\|f_{T,j}^{(i)} - \Phi_{T,j}\right\| \geq tol$
    $\varphi' \leftarrow \varphi' \ \wedge \ \left(\|\Phi_{T,j}\| \leq \mu_j f_{N,j}^{(i)} \ \vee \ \left\|f_{T,j}^{(i)}\right\| < \mu_j f_{N,j}^{(i)}\right)$
  end for
  $sat \leftarrow Satisfy(\varphi')$
end while
$\varphi \leftarrow \varphi \ \wedge \ x^{(i+1)} = R(..., \mathcal{I}_{ex}, x)$ (or $R(..., \mathcal{C}_{ex}, x)$)
• Does the reset create an unsafe solution?
if $Satisfy\left(\varphi^{(i)} \ \wedge \ \neg\phi(x^{(i)})\right)$ **then**
  return $safe = false$
end if
• Does the reset result in solutions that cross guards?
$roots \leftarrow \{\emptyset\}$
$I \leftarrow$ index of edges leaving the computation node $\mathcal{I}$ or $\mathcal{C}$
$J \leftarrow \{\emptyset\}$, $sat \leftarrow true$
while sat **do**
  $sat \leftarrow Satisfy\left(\varphi^{(i)} \ \wedge \ \left(\bigvee_{\forall j \in I} G_j\right) \bigwedge_{\forall k \in J} \neg G_k\right)$
  if sat **then**
    (Denote e as the edge that is crossed)
    $roots \leftarrow roots \cup \varphi$
    $I \leftarrow I \setminus e$
    $J \leftarrow J \cup e$
  end if
end while
Output: safe, roots

A Boolean abstraction of this would be,

$$a \wedge (b \vee c) \wedge (a \vee c),$$

where a, b, c are Boolean variables which are true when their corresponding constraints are true, e.g. $a = true \iff c_1(x) \leq 0$. The Boolean abstraction is then given to a SAT solver to generate a solution. For instance, a, b true and c indeterminate would be a solution which satisfies our example formula.

So far, the solution to our formula is provisional, and only satisfies the Boolean abstraction. We must now determine whether the collection of constraints, corresponding to the Boolean variables with assignments, have real solutions. A set of constraints τ is created from the Boolean abstraction,

$$\tau = \{c_1(x) \circ 0,\ c_2(x) \circ 0\ , ...\},$$

where $\circ = \{=, \leq\}$. Continuing with our example, if the SAT solver has given us the solution a, b true and c indeterminate, then τ consists of the corresponding constraints for a, b,

$$\tau = \{c_1(x) \leq 0,\ c_2(x) \leq 0\}\,.$$

The set of constraints is passed to a constraint solver which determines whether a real solution x exists for the set of constraints in τ, and if it does, returns true. If not, then we can state that the Boolean abstraction does not hold for the real constraints. Thus, we determine that the Boolean solution ν and certain variations of it (the variations are discussed in Section 6.3) are not feasible, and we adapt our original formula accordingly to prevent these combinations from being chosen again.

Algorithm 5 *Satisfy* (Lazy SMT method adapted from [2])

Inputs: ϕ
$\phi' = BooleanAbstraction(\phi)$
while *true* **do**
  $(sat, \nu) \leftarrow SATSolver(\phi')$
  if $sat = false$ **then**
    return $(sat, \nu \leftarrow \emptyset)$
  end if
  $\tau \leftarrow CreateConstraints(\nu, \phi)$
  $sat \leftarrow ConstraintSolver(\tau)$
  if $sat = true$ **then**
    return (sat, ν)
  end if
  $\phi' \leftarrow \phi' \wedge Conflicts(\nu)$
end while
Output: sat, ν

6.1. *Integration of ODEs*

The Lazy SMT solver is required to address CNF formulas with ODE problems as literals. For example, from Algorithm 2 we can have CNF formulas of the form,

$$... \wedge x^{(i+1)} = ODE(f_s, x^{(i)}, dt_d) \wedge ...$$

where f_s is the vector field of a dynamical location, $x^{(i)}$ is some initial state, and dt_d is some time interval. There may be several such literals within the CNF formula, with different vector fields, initial conditions and time intervals. The exact solution for such ODEs can be a difficult problem. Therefore, we include a numerical integration component to the Lazy SMT solver as a pragmatic expedient to resolving these literals.

6.2. *Constraint solutions with maximal feasible set*

The constraint solver ($ConstraintSolver(\tau)$) in Algorithm 5 attempts to find a feasible solution x to the set of constraints in τ. This is an example of a feasibility problem, and can be solved using standard optimisation methods by posing the problem in the following way:

$$\min 0, \ s.t. \quad \begin{array}{c} c_1(x) \circ 0, \\ c_2(x) \circ 0, \\ ... \end{array}$$

However, as it will be discussed in the following section, it is advantageous to establish not only whether it is feasible or infeasible, but also the maximal feasible set. That is, what is the most number of possible constraints that can be simultaneously satisfied. This allows us to consider more sophisticated methods of conflict analysis for when the solution is infeasible, and reduce significantly the size of the problem.

The maximal feasible set can be determined by posing the problem above in the following amended form [3, 4],

$$\min \sum_i^N s_i, \ s.t. \quad \begin{array}{c} c_i(x) - s_i \circ 0, \\ s_i > 0 \end{array} \ , \ i = 1, ..., N.$$

When the above problem is infeasible, the optimal point x will satisfy the largest possible number of constraints, and by complement, violate the smallest number of constraints. Thus, we are able to not only establish the feasibility of the set of constraints, but also, the largest possible subset of constraints, or maximal feasible set, when the problem is deemed infeasible.

6.3. *Conflicting constraints*

Algorithm 5 of the Lazy SMT method has a learning component in the sense that each logical combination of literals ν that is found infeasible – because no set of real variables satisfies the constraints – is appended to the formula ϕ, so that this combination is never repeated in the search for a solution. In the simplest case, the

conflict function appends the formula ϕ each time with the unsuccesful ν,

$$\phi' \leftarrow \phi' \wedge \neg\nu.$$

This approach is inefficient and does not take account of the information that we have at hand. We discussed in the previous section how, by posing the minimisation problem in a certain way, we can maximise the feasible set or minimise the infeasible set. Thus, we can use a more elegant approach which maximises the information returned to the algorithm, and by consequence reduces the search space for a solution to the formula.

Designate I_{feas} as the set of indices of all literals in ν which are feasible, and I_{infeas} as its complement. We know that, if all the constraints with index $i \in I_{feas}$ are true, then all of the constraints in I_{infeas} must not be satisfiable. This is only because I_{feas} contains the indices of the maximal feasible set. If any constraint $i \in I_{infeas}$ were satisfiable, then by definition I_{feas} would not be maximal. The following algorithm incorporates this observation:

for all $i \in I_{infeas}$ **do**
$\phi' \leftarrow \phi' \wedge (\bigvee_{\forall j \in I_{feasible}} \neg\nu_j \vee \neg\nu_i)$
end for

where ν_i is the i^{th} literal in ν. For example, consider the formula,

$$\phi' := a \ \wedge \ b \ \wedge \ c \ \wedge d.$$

We find that a and b are feasible and c and d are not. We thus append our formula with the following:

$$\phi' \leftarrow \phi' \wedge (\neg a \ \vee \neg b \ \vee \neg c)\phi' \leftarrow \phi' \wedge (\neg a \ \vee \neg b \ \vee \neg d).$$

We can further refine the information returned by the conflict function be considering the dependencies between the infeasible and feasible literals. Designate C_{ν_i} as the set of all literals in ν which share at least one real variable in their constraint functions with ν_i. For instance, if ν_1 is the constraint $x_1 + x_2 + x_3 \leq= 0$, then ν_1 consists of the real variables x_1, x_2, x_3, which we express as $\nu_1(x_1, x_2, x_3)$. C_{ν_1} is the set of all literals in ν which contains the variables x_1, x_2, x_3. This could be,

$$C_{\nu_1(x_1,x_2,x_3)} := \{\nu_2(x_1, ...), \nu_3(x_1, ...x_3), \nu_{10}(..., x_2, ...), ...\}\,.$$

Designate $I_{C_{\nu_i}}$ as the indices of all the literals in C_{ν_i} Then, we can modify the above algorithm in the following way:

for all $i \in I_{infeas}$ **do**
$I_{C_{\nu_i}} \leftarrow \{j \ : \ \forall \nu_j \in C_{\nu_i}\}$
$\phi' \leftarrow \phi' \wedge (\bigvee_{\forall j \in I_{C_{\nu_i}}} \neg\nu_j \vee \neg\nu_i)$
end for

This produces much shorter clauses than the previous algorithm, and thus reduces the difficulty of the problem.

6.4. *Convex and concave constraints*

The constraints which appear in the clauses of the formula are not restricted to be convex. In the case of our case study in the next section, both convex and concave constraint surfaces appear regularly in the formulas the Lazy SMT solver is required to solve. This represents a problem. If the Lazy SMT solver returns that the formula is infeasible, how are we to know that this truly is the case? It is possible that the minimisation may have become trapped in a local but infeasible minimum, where a feasible minimum also exists. In which case, our determination of infeasibility is dependent on the initial conditions and the result is unsound. Global optimisation methods are often statistically based, and thus can only assert probable infeasibility.

To solve this problem, we are fortunate in the definition of our MRB hybrid automaton. In Section 2 of the main paper where we described the mechanical properties of the systems treated, we imposed the restriction that the contact surfaces are convex. Consequently, as the constraint surfaces in our formula are derived from these contact surfaces, we limit the types of constraint surfaces to either a convex surface or a concave surface – not both in different regions.

For convex surfaces, infeasibility can be known, and the result is sound. For concave surfaces, it follows that the minimal points are on the edge of the allowable range of the variables in our minimisation problem. Thus, we in effect know how many local minima exist, and approximately where they are, and can cycle through each one by choosing appropriate initial conditions until one becomes feasible. If none are feasible, then we can return the result infeasible – with the minimum number of infeasible constraints – which we now know to be sound.

7. Implementation

DyverseBMC is implemented in Matlab (Mathsworks Inc, USA). Boolean satisfiability problems are solved using Matlab's on-board SAT solver. Numerical integration of ordinary differential equation problems (i.e. $x^{(i+1)} = ODE(f_s, x^{(i)}, T)$ in algorithm 2) is performed by Matlab's ode45 function. Optimisation problems are solved using the NAG toolbox *nag_opt_nlp2_solve* function for minimising arbitrary smooth objective functions subject to smooth linear or nonlinear constraints [5].

The programs that form DyverseBMC and the input files for the examples used in this paper are available at http://staff.cs.manchester.ac.uk/~navarroe/research/dyverse/DyverseRBT_BMC.zip.

8. Limitations

One of the most significant limitations of this approach is the need to use numerical integration in the Lazy-SMT solver to compute ODE functions – that is, the ODE functions added to the CNF formula by algorithm 2. The use of numerical integration forces us to discretise the continuous trajectories into smaller time steps, hidden from us by the ODE solver, to compute each successive step in the procedure outlined in Algorithm 2.

This has two direct consequences. Firstly, it introduces an integration error to our trajectories. This is already a well-established problem in the numerical simulation of rigid-body systems. It leaves us with some doubt about the exact position within

the state-space of each time-step. This integration-error increases as our trajectories grow in time.

The problem becomes particularly significant where a trajectory runs close to violating the safety property. It is possible for instance, that a trajectory will only appear to maintain the property because of the integration-error, whereas if the error was suppressed, it would be found to violate the property. The opposite could also be true: an otherwise safe trajectory could be found to contravene the safety property due to the integration error. A similar argument can be applied for the domain or guard sets. A discrete evolution of the system could be missed because the integration error prevents it from entering a guard set, and vice versa. Our certainty is challenged, and the longer the time interval over which our trajectory evolves, the less certain we become.

The heuristic solution to this problem is to combine the computational demands of using higher-order numerical integrators with a suitably conservative over-bound on the safety property that is larger than the upper-bound of the possible integration error. We know the integration error has an upper-bound as we are only considering trajectories over finite time intervals – we are using *bounded* model checking. In this case, we can at least eliminate the possibility of an unsafe system being declared safe, even if we are unsure of the result.

The second problem is caused by the need to provide a minimum time-step interval in Algorithm 2. It is difficult to decide how coarse this minimum time step can be. Too fine will add to the computational burden of the problem. On the other hand, too coarse can cause other problems. Consider the case where the unsafe region, defined by the complement of the safety property in the state-space, contains a thin strip or appendage. If the minimum step size is too coarse, the discretised trajectory may result in steps on either side of the region but not in the region. The consequence is that that the invasion of the unsafe region – and thus violation of the safety property – will not be registered. A similar occurrence may happen with a guard set. If the trajectory skirts the guard set such that it enters and then exits the guard set over the period of a single time step, then the entering of the guard set will not be registered. This will result in a missed discrete transition of the hybrid automaton.

The composition of the *complete* hybrid automaton prior to the start of the process throws an additional potential problem. Rigid-body problems can contain a large number of possible contact-combinations. This is true even with relatively small rigid-body examples with only a few entities, since each contact-combination merits its own set of locations – each representing the different possible friction and impact states at each contact point – the total number of locations and transitions. This combinatorial explosion means that the size of the hybrid automaton can become substantial even in relatively small models.

Finally, we must consider the complexity of the bounded model checking method. As already described, the method works by building mixed Boolean/arithmetic formulas in CNF formulas which are then solved by the Lazy SMT solver. The formulas broadly correspond to sets of trajectories across the state-space, for instance, a set of trajectories entering a certain guard set will satisfy the corresponding formulas for that guard set. The longer the length of a formula, in terms of the number of clauses and literals, the more difficult it is to find a solution with the consequent longer solving times.

The complexity of solving the formula while exploring dynamical locations (i.e. the procedure outlined in Algorithm 2) increases linearly with the length (in time)

of the trajectories. The number of clauses in the formula does not increase overall while exploring these locations. The only increase in complexity is in the numerical integration of ODEs over longer time intervals. However, the number of clauses in the formula increases significantly when the trajectories include location transitions, particularly transitions to computation nodes. The computation nodes add at least six additional clauses for each entity-to-entity contact and several additional clauses for the computation node guard and domain sets, safety sets, exit conditions, etc. Location transitions also add new clauses to constrain the trajectory hits (or is close to) the boundary of the guard sets. This process is shown in Section 4 and Algorithm 3. This function also adds a new variable, in the form of an elastic time-step, to solve when performing constraint satisfaction in the Lazy SMT solver. This further adds the difficulty of the problem. The increase in number of clauses, and inclusion of new variables makes obtaining satisfiability more difficult and implies that the method is best suited to problems with only a small number of location transitions.

9. A multiple contact problem

We will explore the limitations of DyverseBMC in an example with multiple contacts. Consider an example with three balls spaced as shown in Figure 4. Balls 2 and 3 are attached to the origin by a spring of stiffness k and start off with zero velocity. Ball 1 is in free motion with a velocity of -3 m/sec, that is, it is moving towards the origin. There is no gravity acting on the balls. When released, balls 2 and 3 accelerate towards the centre, while ball 1 moves towards the centre at constant velocity. A multi-impact situation occurs with contact between ball 1 and ball 2, and a second contact between ball 1 and ball 3. We choose a coefficient of restitution of 0.8 for this model. For each ball i, with $i = 1, 2, 3$, we consider the state vector $\mathbf{x}_i = (x_i, y_i, z_i, \alpha_i, \beta_i, \gamma_i)^T$, where x_i, y_i and z_i are positions along their respective axes, and α_i, β_i and γ_i are rotations about the x, y and z axes respectively.

We wish to prove that, for spring stiffness $k = [1, 1.5]$, ball 1 will not gain energy. We determine a gain in energy by observing whether the absolute velocity of ball 1 after impact is greater than before the impact. We can pose the problem formally as,

$$\begin{aligned} \dot{y}_1(t) > 3, \quad & \forall\left(k \in [1, 1.5]\right), \forall\left(\{\mathbf{x}_1(0), \mathbf{x}_2(0), \mathbf{x}_3(0), \dot{\mathbf{x}}_1(0), \dot{\mathbf{x}}_2(0), \dot{\mathbf{x}}_3(0)\} \in I\right), \\ & \forall\left(t \in [0, 0.75]\right), \end{aligned}$$

where t is time in seconds, $\dot{y}_1$ is the velocity of ball 1 in the y-direction, I a set of initial conditions, and the dot denotes derivative with respect to time.

The DyverseRBT tool is used to construct the MRB hybrid automaton for the model and DyverseBMC used to find a counterexample.

First the location where none of the balls are in contact is explored. The process starts by searching over a small time interval, then progressively lengthens the time interval until an event (guard crossing, for example) is triggered. The processing time increases linearly as the time-interval length increases, proportional to the number of discrete steps in the numerical ODE solver. A set of trajectories are found which enter the guard set associated with the edge to an impact node representing a two contact situation. The model checker then explores the impact node, adding new Lagrange multiplier terms until a convergent solution is found. The guard set for the impact node is checked and resets are applied. All trajectories are found to return

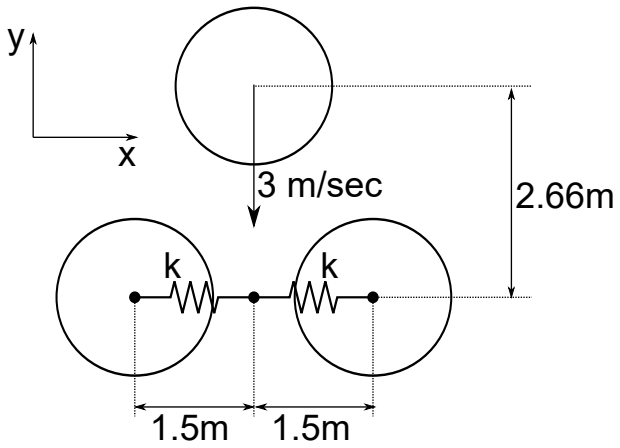


Figure 4. Initial conditions for the three ball problem of Section 9.

to the free-location, which is then explored as before.

The formula for exploring the first dynamical location contains around five clauses with three clauses containing more than one literal. The complexity dramatically increases on a transition. The *root* formula, which constrains trajectories to the boundary of the guard set, adds five clauses to the formula, of which three have more than one literal. Exploring the computation node adds 22 clauses per loop iteration, 16 with more than two literals. If it does not converge then a second loop iteration occurs adding a further 22 clauses, and so on. The result is a significantly more difficult formula for the Lazy SMT to solve.

The analysis of this system took many hours to complete despite the relative triviality of the system, with almost all of this time caused by trying to find trajectories after impact. The length of time involved in finding a counterexample using the approach outlined herein suggests that the DyverseBMC tool in its current form is only suitable for limited multi-contact problems.

References

[1] M. Fränzle, C. Herde, T. Teige, S. Ratschan, and T. Schubert, *Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure*, Journal on Satisfiability, Boolean Modeling and Computation 1 (2007), pp. 209–236.

[2] A. Bauer, M. Leucker, C. Schallhart, and M. Tautschnig, *Don't care in SMT: building flexible yet efficient abstraction/refinement solvers*, International Journal on Software Tools for Technology Transfer 12 (2010), pp. 23–37.

[3] P. Nuzzo, A. Puggelli, S. Seshia, and A. Sangiovanni-Vincentelli, *CalCS: SMT Solving for Non-linear Convex Constraints*, in *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design*, FMCAD 2010, Lugano, Switzerland, 2010, pp. 71–80.

[4] S. Boyd and L. Vandenberghe, *Convex Optimization*, Cambridge University Press, New York, USA, 2004.

[5] The NAG Library, *The Numerical Algorithms Group (NAG), Oxford, United Kingdom*, www.nag.com (2013).

[6] R. Sebastiani, *Lazy satisfiability modulo theories*, Journal on Satisfiability, Boolean Modeling and Computation 3 (2007), pp. 141–224.

[7] C. Studer, *Numerics of Unilateral Contacts and Friction*, Springer-Verlag Heidelberg, 2009.

[8] C. Studer and C. Glocker, *Solving normal cone inclusion problems in contact mechanics by iterative methods*, Journal of System Design and Dynamics 1 (2007), pp. 458–467.

Appendix A. Computation of new contact forces

We briefly review how to compute new contact forces using an iterative Gauss-Seidel relaxation method with proximal point projection, termed the SORPROX method [7, 8]. We will not show the derivation of this method, considering only how to implement it using the notation applied in this paper. The interested reader should consult the reference [7] for the derivation of this procedure.

Consider a system of multiple rigid bodies with P points of contact between them. Designate j as the index of the contact, and $\Phi_{N,j}, \Phi_{T,j}$ as new estimates of the normal and tangential contact forces (or impulses) respectively, at contact j. The remaining notation in this appendix follows the conventions already established in Sections 2 and 3 of the main paper. By direct comparison with the SORPROX method, we can derive the following for impact (impulse) and contact computation nodes:
For an impact computation node $\mathcal{I}_k$ (impact situation), the elements $\Phi_{N,j}\Phi_{T,j}$ are:

$$\Phi_{N,j} := -\mathbf{A}_{j,j}^{-1}\left(\sum_{n=1}^{j-1}\mathbf{A}_{j,n}\lambda_{N,n}^{(i)} + \sum_{n=j+1}^{N_k}\mathbf{A}_{j,n}\lambda_{N,n}^{(i-1)} + \sum_{n=1,n\neq j}^{N_k}\mathbf{B}_{j,n}\lambda_{T,n}^{(i-1)} - (1+\boldsymbol{\epsilon}_{N,j})\dot{g}_{N,j}\right), \tag{A1}$$

$$\Phi_{T,j} := -\mathbf{C}_{j,j}^{-1}\left(\sum_{n=1}^{j-1}\mathbf{C}_{j,n}\lambda_{T,n}^{(i)} + \sum_{n=j+1}^{N_k}\mathbf{C}_{j,n}\lambda_{T,n}^{(i-1)} + \sum_{n=1,n\neq j}^{N_k}\mathbf{B}_{n,j}\lambda_{N,n}^{(i)} - (1+\boldsymbol{\epsilon}_{T,j})\dot{g}_{T,j}\right), \tag{A2}$$

where the superscript (i) indicates the iteration, i.e. the i^{th} iteration, and $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ are obtained from,

$$\begin{pmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{B}^T & \mathbf{C} \end{pmatrix} = \begin{pmatrix} \mathbf{W}_{N,k}^T\mathbf{M}^{-1}\mathbf{W}_{N,k} & \mathbf{W}_{N,k}^T\mathbf{M}^{-1}\mathbf{W}_{T,k} \\ \mathbf{W}_{T,k}^T\mathbf{M}^{-1}\mathbf{W}_{N,k} & \mathbf{W}_{T,k}^T\mathbf{M}^{-1}\mathbf{W}_{T,k} \end{pmatrix},$$

and,

$$\mathbf{W}_{N,k} = (..., W_{N,j}, ...), \quad \mathbf{W}_{T,k} = (..., W_{T,j}, ...), \quad \forall j \in I_k.$$

For a contact computation node $\mathcal{C}_k$ (sustained contact situation), $\Phi_{N,j}, \Phi_{T,j}$ are,

$$\Phi_{N,j} := -\mathbf{A}_{j,j}^{-1} \left(\sum_{n=1}^{j-1} \mathbf{A}_{j,n} \lambda_{N,n}^{(i-1)} + \sum_{n=j+1}^{N_k} \mathbf{A}_{j,n} \lambda_{N,n}^{(i)} + \sum_{n=1, n\neq j}^{N_k} \mathbf{B}_{j,n} \lambda_{T,n}^{(i-1)} + \ddot{g}_{N,j} \right), \tag{A3}$$

$$\Phi_{T,j} := -\mathbf{C}_{j,j}^{-1} \left(\sum_{n=1}^{j-1} \mathbf{C}_{j,n} \lambda_{T,n}^{(i-1)} + \sum_{n=j+1}^{N_k} \mathbf{C}_{j,n} \lambda_{T,n}^{(i)} + \sum_{n=1, n\neq j}^{N_k} \mathbf{B}_{n,j} \lambda_{N,n}^{(i)} + \ddot{g}_{T,j} \right). \tag{A4}$$

Note that we assume here that j is incremented from 1 to P, and that the normal forces are computed before the tangential ones.

The contact forces and impulses must be part of the feasible set implied by the complementarity conditions defined in Section 2. A proximal point projection function is used to enforce this. The function returns the closest point in the feasible set and assigns it to our current iteration of the contact forces. Using the specific contact laws in this paper, we have:

$$\lambda_{N,j}^{(i)} := \begin{cases} \Phi_{N,i} & if \quad \Phi_{N,j} \geq 0, \\ 0 & if \quad \Phi_{N,j} < 0. \end{cases} \tag{A5}$$

$$\lambda_{T,j}^{(i+1)} := \begin{cases} \Phi_{N,j} & if \quad \Phi_{N,j} \geq 0, \\ 0 & if \quad \Phi_{N,j} < 0. \end{cases} \tag{A6}$$

Equations (A1), (A5) and (A6) for an impact computation node, or (A3), (A5) and (A6) for a contact computation node, are iterated until the forces converge:

$$\left| f_{N,j}^{(i)} - f_{N,j}^{(i-1)} \right| < tol \qquad \text{and} \qquad \left\| f_{T,j}^{(i)} - f_{T,j}^{(i-1)} \right\| < tol, \tag{A7}$$

for all P contacts.