Hybrid Simulation Safety: Limbos and Zero Crossings

David Broman

KTH Royal Institute of Technology
Sweden
dbro@kth.se

Abstract. Physical systems can be naturally modeled by combining continuous and discrete models. Such hybrid models may simplify the modeling task of complex system, as well as increase simulation performance. Moreover, modern simulation engines can often efficiently generate simulation traces, but how do we know that the simulation results are correct? If we detect an error, is the error in the model or in the simulation itself? This paper discusses the problem of simulation safety, with the focus on hybrid modeling and simulation. In particular, two key aspects are studied: safe zero-crossing detection and deterministic hybrid event handling. The problems and solutions are discussed and partially implemented in Modelica and Ptolemy II.

Keywords: Modeling, Simulation, Hybrid Semantics, Zero-Crossing Detector

1 Introduction

Modeling is a core activity both within science and engineering. In various domains, there are different kinds of models, such as dynamic models, probabilistic models, software models, and business models. In general, a *model* is an abstraction of something, for instance a process, a system, a behavior, or another model.

Both scientists and engineers make extensive use of models, but for different reasons. As Lee [13] points out, scientists construct models *to understand* the thing being modeled, whereas engineers use models *to construct* what is being modeled. In both cases, the abstraction (the model) contains fewer details than the thing being modeled, which enables the possibility to *analyze* the model. Such analysis can include formal verification, statistical analysis, or simulation.

The latter, *simulation* of models, is the main topic of this paper. Simulation can be seen as a way to perform experiments on a model, instead of experimenting directly on the system or process being modeled [7]. There are many reasons for using modeling and simulation. It can be too dangerous to perform experiments on real systems. It can be cheaper to perform simulations, or the system being modeled might not yet exist.

Regardless of the reason for doing modeling and simulation, it is vital to trust the simulation result to some degree. We say that the *fidelity* of the model is to what extent the model correctly represents the system or processing being modeled. Lee [12, 13] often stresses the distinction between the model and what is being modeled, by giving the famous quote by Golomb [10]: "you will never strike oil by drilling through the map". High model fidelity is necessary, but not sufficient to enable trust of simulation results. To trust the map, as an example of a model, we also need to interpret the map safely. For instance, if an English speaking engineer is using a Russian map to find oil, even a map of high fidelity can lead to incorrect conclusions. Misinterpretations of the map (the model) can result in false positives (drilling through an oil pipe instead of an oil field) or true negatives (drilling through a mine field instead of an oil field). As a consequence, to trust the use of models, not only high model fidelity is needed, but also safe interpretation of the model.

If we make the analogy between a model and a computer program, we can distinguish between two kinds of errors [6]: i) *untrapped errors* that can go unnoticed and then later result in arbitrary incorrect behavior, and ii) *trapped errors* that are handled directly or before they occur. For a computer program written in the C programming language, an array out-of-bound error can lead to memory corruption, where the actual problem can first go unnoticed, and then crashes the system at a later point in time. This is an example of an untrapped error. By contrast, an array out-of-bound error in Java results in a Java exception, which happens directly when it occurs, and makes it possible for the program itself to handle the error. This latter case is an example of a trapped error. A program language where all errors are trapped errors, either detected at compile time using a type system, or at runtime using runtime checks, is said to be a *safe* language.

This paper introduces the idea of making a distinction between safe and unsafe simulations. A *simulation* is said to be *safe* if no untrapped simulation errors occur. A *simulation environment* is said to be safe if no untrapped simulation errors can occur in any simulation. As a consequence, a natural question is then what we mean by *simulation error*. This paper focuses on two kinds of simulation errors that can occur in hybrid modeling languages [2, 5, 14, 15, 16] and cosimulation environments [4, 8]. More specifically, this work concerns both error classification and solution methods. It presents the following main contributions[1]:

- The paper describes two kinds of simulation errors that have traditionally been seen as modeling errors and not as untrapped simulation errors. More specifically, the errors concern i) *unsafe zero-crossing detection*, and ii) *unsafe accidental determinism* (Section 2).
- It describes an approach to make these untrapped simulation errors trapped, by introducing the concept of a *limbo* state. A simulation enters the limbo state when a simulation error is detected. The modeler has the choice of defining the behavior to leave the limbo state in a safe way and continue the simulation, or to terminate the simulation and report the error as a trapped error (Section 3).

[1] All examples in the paper are available here: `http://www.modelyze.org/limbo`

2 Hybrid Simulation Safety Problems

This section describes two problems with hybrid simulation safety. First, it discusses the infamous bouncing-ball problem, where the numerical accuracies of standard zero-crossing detectors make a bouncing ball tunnel through the ground. Second, the section discusses the relations between *accidental* and *intentional nondeterminism*, and the safety problem resulting from *accidental determinism*. The latter problem is illustrated by simultaneous elastic collisions of frictionless balls.

2.1 Unsafe Zero-Crossing Detection

One classic simple example for demonstrating hybrid modeling and simulation is the bouncing ball model. The model demonstrates how a ball is falling towards the ground, and bounces with an inelastic collision, thus bouncing with decreased height. This model can be expressed in any modeling language that supports i) a continuous domain for expressing velocity and acceleration, ii) a construct to numerically detect the collision, and iii) an action statement that changes the sign and magnitude of the velocity of the ball. The following model is a straight forward implementation in the Modelica language:

```
 1 model BouncingBall
 2   Real h,v;
 3   parameter Real c = 0.7;
 4 initial equation
 5   h = 3.0;
 6 equation
 7   der(h) = v;
 8   der(v) = -9.81;
 9   when h <= 0 then
10     reinit(v, -c*pre(v));
11   end when;
12 end BouncingBall;
```

The model is divided into three sections. The first section (lines 2-3) defines the two state variables (`h` for the height of the ball and `v` for the velocity), and one parameter `c` that states the fraction of the momentum that remains after a collision with the ground. The second section (line 5) states an initial equation. In this case, the height of the ball is initiated to value 3. Note that a Modelica tool will implicitly initialize the other variables to zero, in this case the velocity `v`. The third section (lines 7-11) declaratively states the equations that holds during the whole simulation. The `der` operator denotes the derivative of a variable. For instance, `der(h)` is the derivative of the height. Lines 9-11 lists a `when` equation, which is activated when the guard `h <= 0` becomes true. That is, when the ball touches the ground (`h` becomes approximately 0) the `reinit` statement is activated. The `reinit` statement reinitializes state variable `v` to the value of expression `-c*pre(v)`, where `pre(v)` is the left limit value of `v`, before the guard becomes true. Note how the `-c` coefficient both changes the magnitude and the direction of the velocity. Although the bouncing ball example is often used as a “hello world” model for hybrid modeling, it also demonstrates two surprising effects.

Fig. 1(a) shows the simulation result, plotting the height of the ball. As expected, the ball bounces with decreased height until it visually *appears* to sit still, but then suddenly tunnels through the ground. The model demonstrates two phenomena. First, it shows an example of *Zeno behavior*, where infinite number of events (triggering the `when` construct in this case) in finite amount of time. The ball continuous to bounce with lower and lower bounces. Second, the simulation trace shows a tunneling effect, where the ball falls through the ground. Fig. 1(b) shows the last bounces before the tunneling effect. Note how the height of the last bounce is less than 10^{-9} units.

Note, however, that the tunneling effect is not a consequence of the Zeno condition, but of a numerical effect of how traditional zero-crossing detectors detect and handle zero crossings. As can be seen in Fig. 1(b), a zero-crossing detector typically overshoots the crossing slightly, before the action is applied.

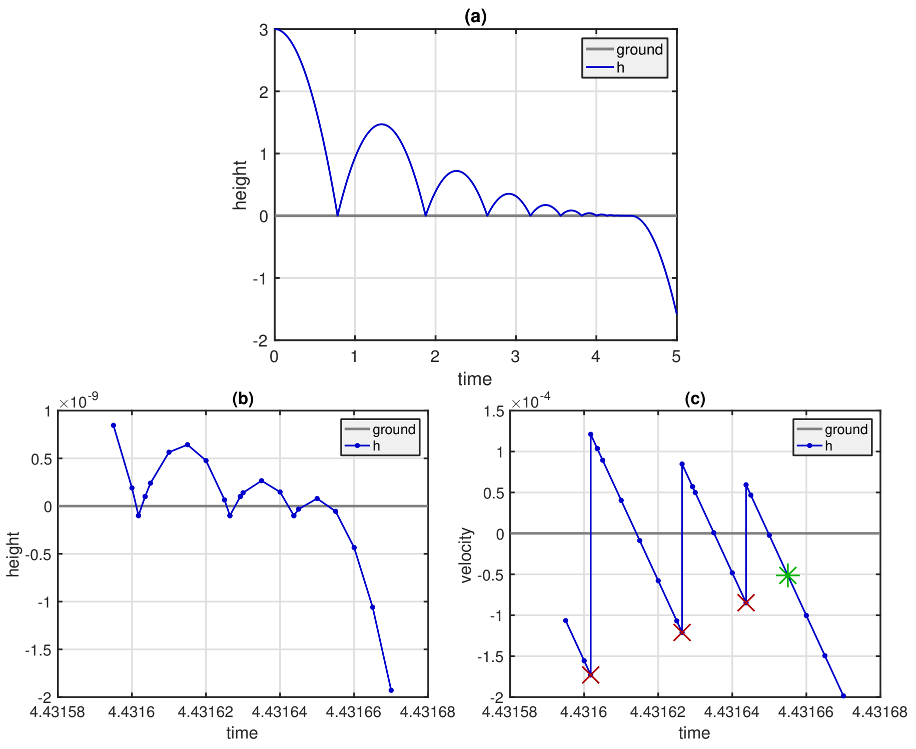

Fig. 1: A bouncing ball that is incorrectly tunneling through the ground. Figure (a) shows the height of the ball during the whole simulation, whereas the two bottom figures zoom in to the tunneling effect, showing the height (b) and the velocity (c). The simulation was performed using OpenModelica v1.11.

To be able to detect a crossing, the bounce needs to get over a certain tolerance threshold, for the crossing to be detected. Fig. 1(c) shows the velocity for the last three bounces. The red crosses mark where zero crossings take place, and where the velocity is changed from positive to negative in the same time instance. At the last instance (marked with a green star) a zero crossing should have been detected, but the bounce has not reached over the tolerance level above zero. Hence, no zero crossing occurs, and the ball tunnels through the ground.

It is important to stress that the Zeno behavior of the model and the numerical tunneling problem are two different things. The former is a property of the model, whereas the latter is a simulation error due to numerical imprecision in a specific simulation tool. The Zeno effect has been extensively studied in the area of hybrid automata, where regularization techniques are used to solve the problem by creating a new model [11]. Traditionally, also the tunneling effect has been seen as a model problem. However, this paper argues the opposite. The tunneling problem is a consequence of an *untrapped simulation error*. A safe simulation environment should handle such problems as *trapped errors*, either by generating an exception state that can be handled within the model, or by terminating the simulation and report an error, before the tunneling effect occurs. A potential solution is discussed in Section 3.

2.2 Unsafe Accidental Determinism

The second problem has been extensively discussed in two recent papers by Lee [12,13]. In these papers, Lee discusses the problem of deterministic behavior of simultaneous events, and illustrates the problem using an example with three colliding balls. This section discusses the problem with the same example, but using Modelica instead of Ptolemy II. The key insight in this section is not the difference in modeling environment, but to view the problem as a simulation safety problem, rather than a modeling problem.

Consider the example in Fig. 2 where ball 1 and ball 3 are moving towards ball 2, which is sitting still. In the example, we assume a frictionless surface and perfectly elastic collision, that is, no energy is lost when the balls collide.
The following Modelica model defines the dynamics of a frictionless elastic ball, with two state variables: x for the horizontal position, and v for the velocity.

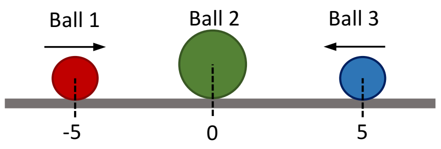

Fig. 2: Illustration of the example with three colliding balls. The balls roll without any friction. Ball 1 and ball 3 move with the same constant speed in opposite directions, where as ball 2 is sitting still before the impact. Note that the problem is a 1-dimensional problem: the balls can only move horizontally, and not in vertical directions.

```
model Ball
  Real x;               // Position state
  Real v;               // Velocity state
  parameter Real x0;    // Initial position
  parameter Real v0;    // Initial velocity
  parameter Real m;     // Mass of the ball
  parameter Real r;     // Radius
initial equation
  x = x0;
  v = v0;
equation
  der(x) = v;      // Relation between position and speed
  der(v) = 0;      // Constant speed, no acceleration
end Ball;
```

For an elastic collision, the momentum and the kinetic energy are preserved.

$$m_1 v_1 + m_2 v_2 = m_1 v_1' + m_2 v_2', \qquad \frac{m_1 v_1^2}{2} + \frac{m_2 v_2^2}{2} = \frac{m_1 {v_1'}^2}{2} + \frac{m_2 {v_2'}^2}{2} \tag{1}$$

Variables v_1 and v_2 represent the velocity before the collision for ball 1 and ball 2, respectively. The velocity after collision is given by v_1' and v_2'. We then have

$$v_1' = \frac{2m_2 v_2 + m_1 v_1 - m_2 v_1}{m_1 + m_2}, \qquad v_2' = \frac{2m_1 v_1 + m_2 v_2 - m_1 v_2}{m_1 + m_2} \tag{2}$$

in the case where $m_1 \neq 0$ and $m_2 \neq 0$. By instantiating the `Ball` model into three components `b1`, `b2`, and `b3`, we get the following model:

```
model ThreeBalls
  Ball b1(x0=-5, v0= 1, r=0.5, m=1);
  Ball b2(x0= 0, v0= 0, r=1.0, m=2);
  Ball b3(x0= 5, v0=-1, r=0.5, m=1);
equation
  //Detecting collision between ball 1 and ball 2
  when b2.x - b1.x <= b1.r + b2.r then
    reinit(b1.v, (2*b2.m*pre(b2.v) + b1.m*pre(b1.v) -
               b2.m*pre(b1.v))/(b1.m + b2.m));
    reinit(b2.v, (2*b1.m*pre(b1.v) + b2.m*pre(b2.v) -
               b1.m*pre(b2.v))/(b1.m + b2.m));
  end when;
  //Detecting collision between ball 2 and ball 3
  when b3.x - b2.x <= b2.r + b3.r then
    reinit(b2.v, (2*b3.m*pre(b3.v) + b2.m*pre(b2.v) -
               b3.m*pre(b2.v))/(b2.m + b3.m));
    reinit(b3.v, (2*b2.m*pre(b2.v) + b3.m*pre(b3.v) -
               b2.m*pre(b3.v))/(b2.m + b3.m));
  end when;
end ThreeBalls;
```

Note that components `b1` (ball 1) and `b3` (ball 3) have the same mass `m=1` and radius `r=0.5`, whereas `b2` (ball 2) has `m=2` and `r=1.0`. The start positions are -5, 0, and 5, for balls 1, 2, and 3, respectively.

The changes in velocity, according to equations (2), are encoded as two `when` equations, each detecting either the collision between ball 1 and 2, or between ball 2 and 3. What is then the expected simulation trace for this model? One expected output might be the plot in Fig 3(a). That is, a simultaneous collision occurs, ball 2 does not move, and the other two balls bounce back with the same velocity. This is actually *not* what happens when simulating the model. Let us take a step back and study the behavior of this model in more detail.

Consider Fig 3(b) and Fig 3(c). These two simulation traces show the same example as above, with the difference that in Fig 3(b), ball 1 starts a bit closer to the middle ball, whereas in Fig 3(c) ball 3 starts a bit closer. As expected, in the first case, ball 1 hits ball 2 first, that makes ball 1 bounce back (it is the lighter of the two) and ball 2 starts to move towards ball 3. Then ball 2 hits ball 3, which bounces back and ball 2 changes direction again. In the first case, ball 3 bounces back at a higher speed because the middle ball's energy from the first hit gives ball 3 the extra speed. As expected, Fig 3(c) shows the reverse, when ball 3 hits ball 2 first.

Now, imagine that the distances between ball 1 and the middle ball, and ball 3 and the middle ball get closer and closer to equal. As long as one of the balls hits first, this will affect the other ball. Hence, the limit for the two cases are not the same. As Lee [13] points out, the model may be seen as nondeterministic in the case when both the balls collide simultaneously. In that case, either ball 1 or ball 3 hits the middle ball first, but nothing in the model indicates the order. Again, the model is nondeterministic in this specific point.

In the previous two plots, the distances between the balls were not the same. Fig 3(d) shows the actual simulation result when simulating `ThreeBalls` where the distances between the balls are equal. We get a simulation trace, but is it the correct one? Obviously no. We can notice two things. First, even if ball 1 and ball 3 arrive at the same speed from the same distances to the middle ball, ball 2 moves to the left after impact. Why is the ball moving in that direction and not the opposite direction? Second, note how ball 1 and 2 tunnel through each other, and are at the same position at time 8 (which should be physically impossible). The reason ball 2 moves to the left is that both `when` equations are activated simultaneously and that the code within the two `when` blocks (lines 8-11 and lines 15-18) are executed in the order that they are stated in the model. Hence, the velocity for ball 2 is initialized twice (lines 10 and 15), where the last one (line 15) gives the final result.

To make the situation even worse, assume that we switch the order of the two `when` equations in model `ThreeBalls`, that is, the `when` equation for detecting collisions between ball 2 and 3 comes before the `when` equation for detecting collisions between ball 1 and 2. Modelica is a declarative language, where the order of equations should not matter. Hence, we might expect to get the same incorrect result. Unfortunately, this is not the case. The simulation

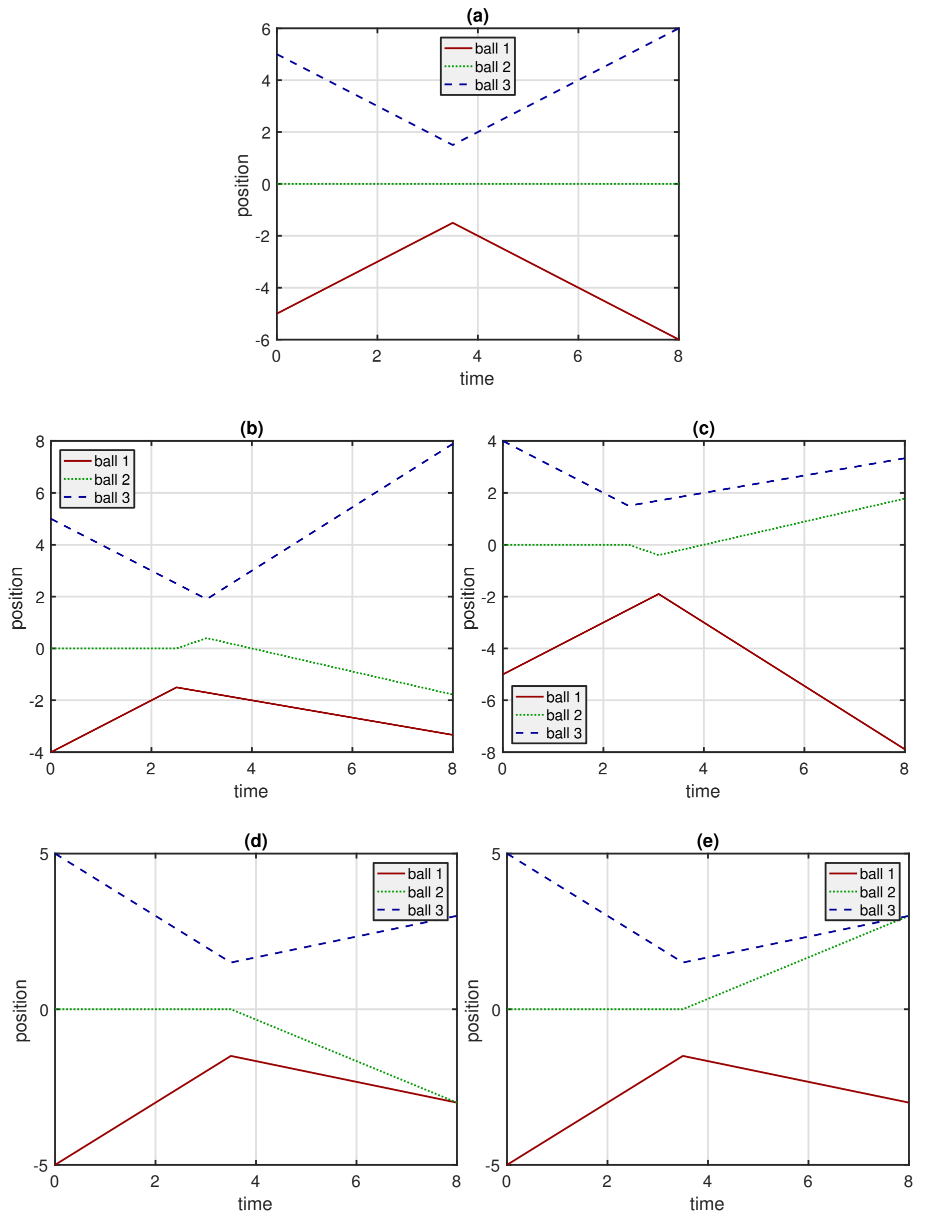

Fig. 3: Simulation cases for the `ThreeBalls` model. Figure (a) shows an ideal result when the balls have the same initial distances. Figures (b) and (c) show simulation traces, where initial distances between the balls are not equal. Figures (d) and (e) show unexpected simulation results, where the initial distances between the balls are the same, but where the `when` blocks have different order.

result for the new model, where the `when` equations have switched order, is shown in Fig 3(e). Note how ball 2 moves in the opposite direction after impact because the order of the impact from ball 1 and 3 has changed.

Lee [13, p. 3:24] argues, based on a similar example, as follows: "It would probably be wise to assume that determinism is incomplete for any modeling framework that is rich enough to help design an understand CPS, where discrete and continuous behaviors inevitably mix.". If the order of the evaluation of `when` equations matter and the order is left unspecified, the model is indeed nondeterministic: there are two possible interpretations. However, this paper argues that it is important to not mix the two separate issues of the determinism of the model, and the determinism of the simulation.

Fig. 4 shows a matrix, where we introduce the concepts of intensional determinism/nondeterminism, and accidental determinism/nondeterminism. *Intensional determinism (ID)* for a modeling and simulation environment is typically what is intended in many simulation environments for cyber-physical systems (CPS). ID means that the simulation of deterministic models yields deterministic simulation results. The same model simulated with the same input always results in the same simulation result. *Intensional nondeterminism (IND)* means that the model itself is nondeterministic, and that the simulator may use random samples to generate the simulation result. Monte Carlo methods fall within this category. Many useful formalisms, languages, and environments fall within the categories of ID and IND.

The accidental categories are more problematic. *Accidental nondeterminism (AND)* is when a simulator for a deterministic model generates different simulation traces, even if the same model with the same input is used. If a simulator behaves within the AND-category, it typically means that there is an error in the simulator. For instance, if a simulator is incorrectly using a multithreaded execution environment, where the simulation result depends on the thread interleaving, the simulator might give different results for different executions.

The last category, *accidental determinism (AD)* is the one that is particular interesting in this example. In this case, a nondeterministic model always yields

	Determinism	**Nondeterminism**
Intensional	**ID** *"Deterministic model with a deterministic simulation result"*	**IND** *"Nondeterministic model with random choice during simulation"*
Accidental	**AD** *"Nondeterministic model with a deterministic simulation result"*	**AND** *"A deterministic model that results in different simulation results for different executions"*

Fig. 4: A matrix that shows the relationship between intensional determinism/nondeterminism and accidental determinism/nondeterminism.

the same simulation result. This is exactly what happens in our simulation example of the `ThreeBalls` model. From Fig 3(b) and Fig 3(c) we know that the order in which balls 1 and 3 hit ball 2 has a direct implication on the simulation result. When the distance between the balls is the same, both `when` equations are activated simultaneously. The Modelica tool then decides on an evaluation order for the equations. If the `reinit` statements were independent of each other, the order would not matter. However, in this case, the order matters. As it turns out, the simulator (OpenModelica v1.11 [9]) executes the constructs in linear order, which is the reason for the different simulation traces for Fig 3(d) and Fig 3(e). We have an accidental deterministic behavior, where the original model was nondeterministic, but where the simulation result is deterministic. Recall that the actual activation choice is made on the order the `when` equations are defined in the file. Accidental determinism is an example of unsafe simulation: the error is untrapped, that is, we get a simulation result without warnings, even though the result itself is not deterministic.

3 Safe Simulations using the Limbo State

The previous section showed two examples of unsafe simulation behavior. In both cases, the simulation continued and produced a result, without giving any errors or warnings. These are examples of untrapped simulation errors. Although an error occurs at a specific point in time (the tunneling effect or incorrect collision), the simulator still produces a simulation result. The purpose of this section is to illustrate the idea of how to make the untrapped errors trapped, thus enabling safe simulations.

3.1 The Limbo State

The key idea is to introduce three conceptual states in a simulator: i) the *safe* state, ii) the *limbo* state, and iii) the *unsafe* state. During simulation, the simulator is in one of these three states. Note that these are states of *the simulator* itself, and not modes in a specific model. The idea of the limbo state is first described abstractly, followed by a concrete discussion in the context of the previous two problem examples.

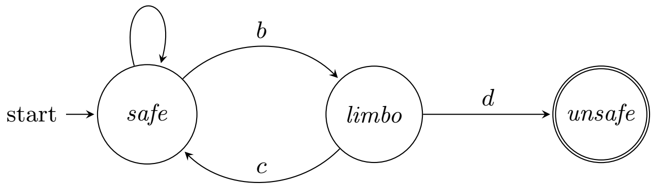

Fig. 5: A finite state machine diagram that includes the limbo state.

Fig. 5 depicts a finite state diagram with the three states. A simulation starts in the safe state. If no errors occur, the simulator stays in the safe state. If a potential error occurs, transition b is taken to the limbo state. The limbo state means that the simulator is in between a safe and an unsafe state. The error *can potentially happen*, but has not yet taken place. From the limbo state, either the simulation is safely terminated with an error message (a trapped error), or transition c is taken back to the safe state. It is the modeler's responsibility to augment the model, such that transition c can be taken. If the error occurs in the limbo state, transition d is taken. If the simulator is safe, transition d should happen *when the error occurs*, that is, it should terminate the simulation at the simulation time of the error. Thus, transition d should generate a trapped error, indicating that the simulation reached an unsafe state at a specific point in time.

The reader might now ask why we see the described problems as simulation errors, when the user still can modify the model to avoid the error? Is it not a modeling error then? The point is, again, that the errors which appear during simulation must be trapped. However, the same model can still be valid for different simulation input. For instance, the bouncing ball model in Figure 1(a) is valid before time 4, since the simulation error happens sometime between time 4 and 5. Let us now consider the two problems in Section 2 in turn.

3.2 Safe Zero-Crossing Detector

The tunneling problem described in Section 2.1 can easily be detected using multiple levels of zero-crossings [17]. The problem with traditional zero-crossing detectors, such as the `when` equations in Modelica and level-crossing detector actors in Ptolemy II, is that they can easily be used in an unsafe way. The key idea is instead that a modeling language should *only* provide safe zero-crossing detectors, where the tunneling effect cannot occur.

Fig. 6 depicts the structure of a safe zero-crossing detector. A safe zero-crossing detector has a safe region, a limbo region, and an unsafe region. The detector consists of three levels of detection mechanisms: i) *zero level* that detects the actual zero crossing, ii) *limbo level* that detects when the limbo region is entered, and iii) the *unsafe level*, which detects that the model did not leave the limbo state correctly.

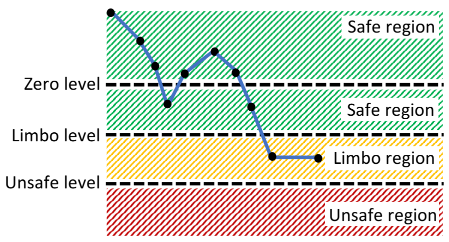

Fig. 6: The three different crossing detection levels and regions.

Returning to the state machine in Fig. 5. Transition *a* is taken each time the *zero level* is crossed. That is, the simulation is still safe, even if the zero level is crossed. In the bouncing ball example, this happens every time the ball bounces correctly (see the example trajectory line in Fig. 6). Transition *b* is taken if the variable value crosses the limbo level. In the bouncing ball example, this occurs when the ball is starting to tunnel through the ground. Note that this does not have to be an error. If the modeler detects the tunneling effect, and then changes the mode of the ball to stay still (no acceleration or velocity), the simulation changes state to be safe again (transition *c*), or stays in the limbo region (see again the example trajectory line in Fig. 6). However, if the model is incorrectly implemented, as in the example in Section 2.1, the unsafe level will be crossed. In such a case, the simulation environment should generate a trapped error, by terminating the simulation and by reporting the simulation time of the error. A safe modeling and simulation language should only include safe zero-crossing detectors as primitives, making it impossible to use unsafe zero-crossing detection. Consider now the following Modelica model.

```
model SafeBouncingBallFinal
  Real h,v;
  discrete Real a(start = -9.81);
  parameter Real c = 0.7;
  parameter Real epsilon = 1e-8;
  Boolean limbo;
initial equation
  h = 3.0;
  v = 0;
  limbo = false;
equation
  der(h) = v;
  der(v) = a;
  when h <= 0 then
    reinit(v, -c*pre(v));
  end when;

  //Limbo state action
  when limbo then
    reinit(v,0);
    a = 0;
  end when;
  // Detecting limbo level
  when h <= -epsilon then
    limbo = true;
  end when;
  // Detecting unsafe level
  when h <= -2*epsilon then
    terminate("Unsafe Zero Crossing");
  end when;
end SafeBouncingBallFinal;
```

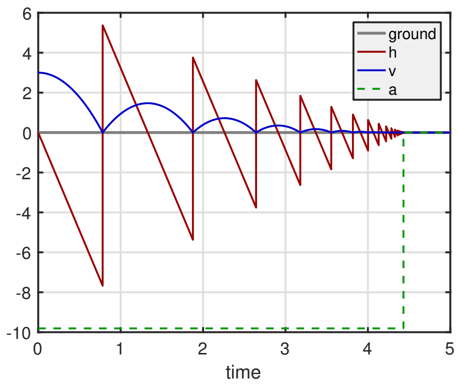

Fig. 7: A safe bouncing ball that stays on the ground. Note how the acceleration `a` of the ball transitions from -9.81 to 0 when the ball comes to rest.

Fig. 7 shows the simulation trace of simulating the model. A few remarks are worth making. We can see that the `when` equation for detecting the zero crossing is unchanged compared to the previous section. What has been added are two more `when` equations that detect the limbo level (line 24), and the unsafe level (line 28). If the limbo region is entered (line 24) a boolean variable is updated, which triggers the limbo action (lines 19-22), where the ball is put to rest. Note that if we reach the unsafe region (line 29), the simulation is terminated. In this case, this is done explicitly in the model, but an ideal modeling language should include such detection automatically. It is not obvious how to extend Modelica in this way, but an interesting direction is to be able to specify invariants of safe states, as done with invariants in hybrid automata [1].

A zero-crossing detector can be generalized into a directional level-crossing detector, that can detect arbitrary level in one direction. Fig. 8 shows a simple safe level-crossing detector that is implemented as an actor in Ptolemy II. The main model called *Bouncing Ball* is a modified version of the bouncing ball example from [12]. Two changes have been made: i) the `Ball` model has been replaced with a `ModalBallModel`, and ii) the original level-crossing detector has been replaced with a new safe level-crossing detector. Note that the safe level-crossing detector actor has approximately the same interface as the level-crossing detector in the Ptolemy II standard library, with the main difference that it also has an output port called `limbo`. The safe level-crossing detector outputs a discrete event on the `limbo` port if it detects a limbo state. When it is used in the bouncing ball example, it means that the ball is just about to start to tunnel. In the example model, the limbo port is connected to the `ModalBallModel` actor's `stop` port. The modal model has two modes, i) the ball is `falling`, and ii) the ball is sitting `still`. If the modeler forgets to connect the `limbo` port, the safe level-crossing detector reports a trapped error.

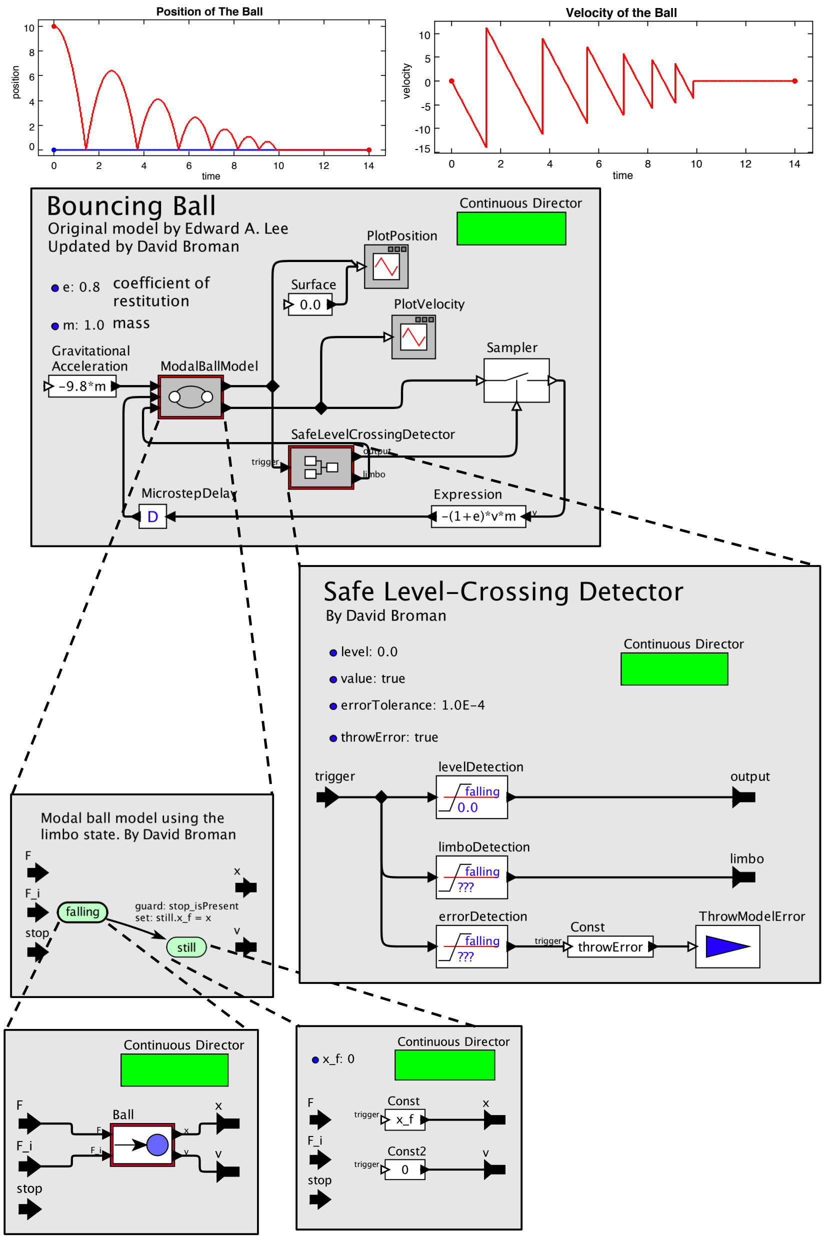

Fig. 8: An implementation of a safe level-crossing detector in Ptolemy II. The original open source model, before extending it with the safe level-crossing detector, is available here: `http://ptolemy.org/constructive/models`
.

3.3 Safe Deterministic Event Handling

In the colliding ball example in Section 2.2, the root of the accidental deterministic behavior was simultaneous events. It is extremely hard (if not impossible) to guarantee that events cannot happen simultaneously. Numerical imprecision, both due to round-off errors and integration errors, makes it hard to give any guarantees. A naive solution would be to always enforce that no events occur simultaneously by arbitrarily selecting an order. However, this will lead to the problem of accidental determinism. If the order actually matters, such arbitrary deterministic choice would result in an unsafe behavior.

Instead, our proposal is to again make use of the limbo state diagram, as shown in Fig. 5. The transition b should be activated when two events are sufficiently close to each other. The exact meaning of *sufficiently close to each other* can be configured using a numerical tolerance level. This means that a model will transition into the limbo state when simultaneous events occur. This does not have to be an error. If the modeler knows how to handle the specific case, he/she can express this in the model (assuming that the modeling language is expressive enough) and then make a transition back to the safe state. If no such case for simultaneous event is implemented, the simulation tool must report a trapped error. In Modelica, `elsewhen` constructs can be used to implement such special cases. This is indeed what was done to create the simulation plot in Fig 3(a). Note that a nondeterministic model with missing cases can be seen as an underspecified model. By adding all missing cases and completely specifying the model, we convert a nondeterministic model into a deterministic model.

4 Conclusions

This paper presents and discusses the idea of safe simulation. In particular, it makes a distinction between trapped and untrapped errors. As part of the solution, the notion of a limbo state is introduced. The preliminary work is illustrated using small examples in Modelica and Ptolemy II. However, to make the approach useful in practice, the safety concepts need to be integrated as explicit parts of a modeling language and a simulation environment. An interesting direction for future work is to investigate if type systems in modeling languages [2,3] can be used to statically detect and eliminate untrapped errors.

Acknowledgments

I would like to thank Edward for the really great collaboration we have had over the years. And, congratulations on your birthday! I would also like to thank Cludio Gomes, Oscar Eriksson, and the anonymous reviewers for many useful comments. Finally, I would like to acknowledge and thank Bernhard Rumpe for pointing out the connection between nondeterminism in models and underspecification.

This project is financially supported by the Swedish Foundation for Strategic Research (FFL15-0032).

References

1. Rajeev Alur, Costas Courcoubetis, Thomas A Henzinger, and Pei-Hsin Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In *Hybrid systems*, pages 209–229. Springer, 1993.
2. Timothy Bourke and Marc Pouzet. Zélus: a synchronous language with ODEs. In *Proceedings of the 16th International Conference on Hybrid Systems: Computation and Control (HSCC)*, pages 113–118. ACM, 2013.
3. David Broman, Peter Fritzson, and Sébastien Furic. Types in the Modelica Language. In *Proceedings of the Fifth International Modelica Conference*, pages 303–315, Vienna, Austria, 2006.
4. David Broman, Lev Greenberg, Edward A. Lee, Michael Masin, Stavros Tripakis, and Michael Wetter. Requirements for hybrid cosimulation standards. In *Proceedings of 18th ACM International Conference on Hybrid Systems: Computation and Control (HSCC)*, pages 179–188. ACM, 2015.
5. David Broman and Jeremy G. Siek. Modelyze: a gradually typed host language for embedding equation-based modeling languages. Technical Report UCB/EECS-2012-173, EECS Department, University of California, Berkeley, June 2012.
6. Luca Cardelli. Type Systems. In *The Computer Science and Engineering Handbook*, chapter 97. Second edition, 2004.
7. François E. Cellier. *Continuous System Modeling.* Springer-Verlag, New York, USA, 1991.
8. Fabio Cremona, Marten Lohstroh, David Broman, Edward A. Lee, Michael Masin, and Stavros Tripakis. Hybrid co-simulation: it's about time. *Software & Systems Modeling*, 2017.
9. Peter Fritzson, Peter Aronsson, Håkan Lundvall, Kaj Nyström, Adrian Pop, Levon Saldamli, and David Broman. The OpenModelica Modeling, Simulation, and Software Development Environment. *Simulation News Europe*, 15(44/45):8–16, 2005.
10. Solomon W Golomb. Mathematical models: Uses and limitations. *IEEE Transactions on Reliability*, 20(3):130–131, 1971.
11. Karl Henrik Johansson, Magnus Egerstedt, John Lygeros, and Shankar Sastry. On the regularization of Zeno hybrid automata. *Systems & control letters*, 38(3):141–150, 1999.
12. Edward A Lee. Constructive models of discrete and continuous physical phenomena. *IEEE Access*, 2:797–821, 2014.
13. Edward A Lee. Fundamental Limits of Cyber-Physical Systems Modeling. *ACM Transactions on Cyber-Physical Systems*, 1(1):3, 2016.
14. Modelica Association. *Modelica - A Unified Object-Oriented Language for Physical Systems Modeling - Language Specification Version 3.4*, 2017. Available from: `http://www.modelica.org`.
15. Henrik Nilsson, John Peterson, and Paul Hudak. Functional Hybrid Modeling. In *Practical Aspects of Declarative Languages : 5th International Symposium, PADL 2003*, volume 2562 of *LNCS*, pages 376–390, New Orleans, Lousiana, USA, January 2003. Springer-Verlag.
16. Claudius Ptolemaeus, editor. *System Design, Modeling, and Simulation using Ptolemy II.* Ptolemy.org, 2014.
17. Michael M. Tiller. *Modelica by Example.* Online book, 2017. Available at `http://book.xogeny.com/`. Last accessed: Sep 9, 2017.