The Hypercubes abstract domain for physics simulations

Giulia Costantini[1], Pietro Ferrara[2], Agostino Cortesi[1], and Giuseppe Maggiore[3]

[1] University Ca' Foscari of Venice, Italy
{costantini,cortesi}@dsi.unive.it
[2] ETH Zurich, Switzerland
{pietro.ferrara}@inf.ethz.ch
[3] NHTV University of Breda
{maggiore.g}@nhtv.nl

Abstract. It is difficult to statically verify interesting properties of physics simulations. Physics simulations are widespread (games, ballistics, weather, etc.), complex to build and even more complex to test thoroughly. In this paper we propose a novel technique based on abstract interpretation that is able to track relevant information on the behaviour of physics simulations. Thanks to our technique, properties which would require extensive testing to be verified can be proven statically, thereby reducing the cost of building physics simulations.

1 Introduction

It is difficult to statically verify interesting properties of physics simulations. The difficulty arises from the particular shape of programs simulating physics behaviours, because such programs feature: (i) a while loop which goes on "forever"; (ii) a complex state made up by multiple floating point variables; (iii) strong dependencies between variables. We now explain with more details these points:

1. a simulation is essentially an initialization of the state (i.e., the variables which compose the simulated world) followed by a `while` loop which computes the numerical integration over time (i.e., the inductive step of the simulation). This `while` loop may either be executed in real-time (for example, in a game) or be computed off-line (for example, weather forecasts). The condition of the while loop is `true` for the whole duration of the program (because the program stops when the condition becomes `false`), so we cannot know beforehand the number of loop iterations which will be executed by the program.
2. the variables of a physics simulation are floating point, because they represent continuous values that generally map directly to physical aspects of the real world. For example, simulations deal with positions, velocities, accelerations, amount of fuel left, and so on.

3. the variables of a simulation are strongly inter-related, because often the simulation makes decisions based on the values of particular variables. For example, the velocity of an object changes abruptly when there is a collision (which depends on the position of the object). Similarly, the position changes accordingly to the velocity, which in turn depends on the acceleration which may derive from the position (for a gravitational field) or from other parameters. Losing track of these inter-relationships can have disastrous consequences for the analysis: precision is lost quickly so that no static property can be verified.

Physics simulations are used in a lot of fields for very important applications. Think about military applications (ballistics), or aero-spatial simulations (NASA, air-planes, etc.), or even weather forecasts and environmental simulations. Moreover, such simulations are widely used in games, which are important from an economic point of view.

Depending on the application, there may be multiple kinds of properties which need to be verified in order for the application to work correctly. Usually, these properties have to do with *certainly* reaching (or not reaching) given configurations of the state. For example, we may want to ensure that a rocket reaches a stable orbit instead of falling to the ground, the projectile hits its target, and so on. We discuss in depth a case study in Section 2.

Traditional approaches to static analysis do not perform well with this kind of programs. Then, to validate such programs, we would need extensive testing, given the huge amount of possible values each variable can assume, a problem exacerbated by the presence of multiple such variables. Losing even one possible trace of execution can lead to disastrous consequences such as [**?**]. We then need stronger guarantees for these programs. To give such guarantees, domain-specific approaches are needed.

Problem statement We want to build a technique to statically analyse programs which encode physics simulations. Our technique must satisfy the following requirements:

- *genericity*, with respect to the specific simulation;
- *performance*, i.e. not being too slow to be computed;
- *precision*, i.e. being able to prove interesting properties of the program

To create this technique, we build on the abstract interpretation framework. In particular, we keep the basic idea of an abstract domain to represent the program data and abstract operations to approximately "execute" the program statements. The novelty lies in the fact that the values of our abstract domain (i.e., abstract states) approximate *all* the variables of the program at the same time. Usually, we associate an abstract state to each variable at each point of the execution. Instead, in our technique, an abstract state tracks all the variables *together*. We achieve this through the use of sets of hyper-cubes (see Sections 3 and 4).

Our technique is easy to implement in a functional framework (see Section 5), greatly reducing the time to integrate it and use it in practice. Our initial benchmarks (see Section 6) show promising results in terms both of performance and precision in proving properties which would otherwise require extensive testing. Also, our technique is sufficiently generic to be successfully applied to other kinds of programs: an example is presented in Section 7.

As a final note, we observe that our approach offers plenty of possible venues in order to improve its results. In particular, in Section 10 we discuss:

- how to increase precision by intersecting our hyper-cubes with arbitrary bounding volumes that restrict the relationships between variables
- how to improve performance by using an adaptive subdivision scheme to have hyper-cubes of differing sizes
- the study of the derivative with respect to time of the iterations of the main loop in order to define temporal trends

Thanks to our technique, properties which would require extensive testing (like the ones listed at the beginning of this Section, i.e. ensuring that a rocket does not fall to the ground and so on) can be proven statically. Removing the need of extensive testing reduces greatly the cost of building phisics simulations. As an informal proof of the possible impact of this direction of studies consider how much time and effort are spent on testing in the industry of videogame development [**?**], which is a huge industry [**?**]. We argue that, by using our abstract interpretation framework, not only many of these tests would be rendered useless, but also the assurances obtained would be much stronger, since the proven properties are valid for *every* execution of the program.

2 The case study of bouncing balls

To test our technique, we chose a case study that offers a challenging analysis environment.

We consider a program which discreetly generates bouncing balls on the screen. The interval between the creation of two balls is constant and known at compile time (let us call it `creationInterval`). The balls all start at the top left corner of the screen (in a fixed position), but each one has a different initial velocity (generated randomly at each ball creation). However, the horizontal direction of the ball is always towards the right of the screen. Whenever a ball touches the "ground" (i.e., the bottom of the screen), it bounces (i.e., its vertical velocity is inverted). When a ball reaches the right border of the screen, it is removed from the world. For performance reasons, we cannot allow more than a certain number N of balls to be on the screen at the same time, since each ball requires computations for its rendering and updating. Thus, what we would like to prove is that *there never are more than N balls on the screen at the same time*. Since we know that we generate a ball each `creationInterval` seconds, this property can be modified into verifying that, $N \times$ `creationInterval` seconds after the generation of a ball, such ball has already exited from the screen. In

fact, we know that after such time ($N \times$ `creationInterval` seconds), other N balls will have been generated, so the first ball must have already exited. [4] To verify this property (which we call *Property 1*), we have to consider the program which simulates the creation and movement of a single ball. We reported the code of such program in Listing 1.1. We used a functional syntax for the code, since we implemented the analyser in a functional framework (see Section 5 for more details).

Listing 1.1. Case study code

```
let px = 5.0f;
let py = 40.0f;
let vx = rand(10.0f, 15.0f);
let vy = rand(-10.0f, 10.0f);
let dt = 0.06f;

while(true) {
   if( py >= 0.0f )
   {
     (px, py) = (px + vx * dt, py + vy * dt)
     (vx, vy) = (vx, vy) * 0.8f
   } else {
     (px, py) = (px + vx * dt, 0.0f)
     (vx, vy) = (vx, -vy) * 0.8f
   }
}
```

The structure of this program respects the generic structure of a physics simulation, as explained in Section 1. In fact, it starts with the world initialization, that is, the assignment of the initial values to the program variables. The variables of this program are the following:

- (`px`, `py`) which represent the current position of the ball. Intuitively, `px` represents its horizontal position on the screen, while `py` represents its vertical position (we work in two dimensions). The ball is always generated in the same place, since the initial position is constant. In particular, each ball is generated at the top left corner of the screen. Note that the x-coordinate increases towards the right of the screen, while the y-coordinate increases towards the top of the screen: see Figure 1 for an example of screen coordinates.
- (`vx`, `vy`) which represent the current velocity of the ball. In particular, `vx` represents the horizontal velocity, while `py` represents its vertical one. The velocity is generated randomly. Note that the horizontal velocity is always

[4] This property is more restrictive than the original one, since the balls do not necessarily exit the screen in the same order they were created, depending on their random initial velocity. Anyway, the increased strictness means that, if we prove this property, we have also proven the original one.

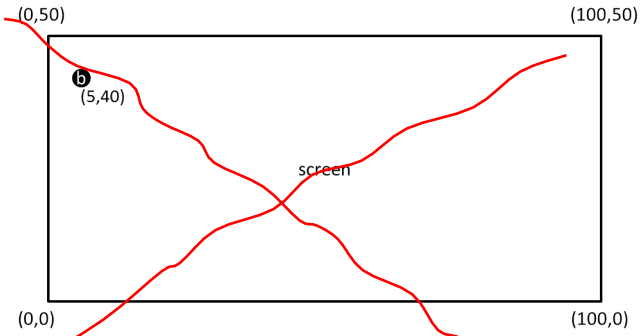


Fig. 1. An example of screen dimensions. The black dot corresponds to the point of ball generation

positive, because we want to throw the balls towards the right of the screen (where they will exit the world). The vertical velocity, instead, can be positive or negative.
– `dt` which represents the time interval between iterations of the loop. This value is constant and known at compile time.

The while loop updates the world variables, that is it updates the ball position (depending on the current velocity) and the ball velocity (depending on eventual bounces and on gravity acceleration). In particular, the loop is structured as follows:

– if the ball is not yet below the ground (i.e., its vertical position is greater or equal to 0.0f), its position is updated according to the rule of uniform linear motion. The velocity is also updated, multiplying it by a constant factor of 0.8f (which represents .... what? attrito? ma la velocitá verticale non dovrebbe aumentare per la gravita'?).
– otherwise, the ball has touched the ground and it must bounce. To do this, we update the horizontal position as usual, but we force the vertical position to zero (it cannot go underground). Moreover, the vertical velocity is inverted (instead of going down, the ball must go up) and decreased (along with the horizontal velocity) through the same constant factor used before (0.8f).

The interest of this case study lies in the fact that "traditional" approaches (which do not consider relationships between variables) are not able to verify *Property 1*. Consider for example the Interval domain. Given the wide range of the possible initial velocities, after a few iterations of the while loop, the intervals representing the possible ball positions (both in x and y) become very large: precision is lost quickly. Moreover, as soon as the intervals become wider, the analysis is no more able to choose which branch of the `if − then − else` to take: this speeds up the precision loss, which makes the analysis end with no interesting information on the problem. See also Section 6 for the experimental

results of the Interval domain (together with those of the Hypercubes domain) in our case study.

3 The Hypercubes domain

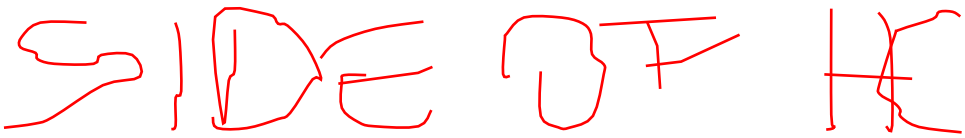
In this Section we are going to present our abstract domain, $\mathcal{H}$, first intuitively and then formally.

As hinted in Section 1, we will slightly modify the traditional abstract interpretation framework. Usually, an abstract domain is defined to approximate the values of single variables: this means that each variable is associated to a single element of the abstract domain (i.e. to an *abstract state*, which, in turn, represents a set of concrete values), so that, at each point of the program, we have an abstract state for each variable. Each statement of the program is executed with the abstract semantics associated to the domain, thus modifying the abstract states of variables throughout the program. For example, when analysing a program with three integer variables, we always deal with three abstract states (for example, three intervals if we use the Interval domain, or three signs if we use the Sign domain, etc.). In our approach, instead, an abstract state approximates *all* the variables of the program together. This means that, at each point of the program, we just have a single abstract state instead of one for each variable. We call this the *Global Abstract State* of the program (abbreviated with $\mathcal{GAS}$). In the previous example (a program with three integer variables), the global abstract state approximates all the three variables together.

Intuitively, our global abstract state tracks a relational and disjunctive kind of information: the $\mathcal{GAS}$ is a set of vectors, and each vector contains one abstract value (coming from some other abstract domain) for each variable of the program. Each vector represents a set of admissible combinations of values for all variables. The information tracked with the $\mathcal{GAS}$ is *relational* because the abstract value of a variable coming from a certain vector v is admissible *only* in combination with the abstract values of the other variables in the same vector v. The information is also *disjunctive*, because the concretization of a $\mathcal{GAS}$ is the union of all its vectors concretizations. The concretization of a single vector is the combination of all the concretizations of elements in the vector (i.e., the concretizations of each program variable through its abstract value in the vector).

In this way, we can track relational information between variables of the programs. Also, this representation will be fundamental to lose as little precision as possible when dealing with branches, as we will explain later.

Now we are going to explain in more detail how a vector of our $\mathcal{GAS}$ is made. Remember that the $\mathcal{GAS}$ is a *set* of such vectors. Each vector has a cardinality equal to the number of program variables: this means that each variable is associated to a precise position of the vector (and it is the same in all vectors). In the example cited before (a program with three integer variables), the vectors of the $\mathcal{GAS}$ would be vectors made by three elements. In particular, the three elements would be three abstract values, one for each variable. We can



choose which abstract domain to use to represent single variables depending on the kind of application, and also on the particular variable. We are currently focusing on physics simulations, and (as we said in Section 1) the variables of physics simulations are, generally, floating point numbers. Then, we choose to abstract each floating point variable with an interval of floating point values. To create a more compact and light notation for our vectors, though, we consider intervals of fixed width and we do not store the specific interval range but a single integer representing it. In particular, each variable x_i is associated to an interval width (specific only for that variable), which we call w_i and is a parameter of the analysis. In Section X you can find a discussion on how to choose the interval widths for each variable; for now, just notice that, the smaller the width of a variable is, the more granular and precise the analysis is with respect to that variable (and the more computationally heavy the analysis is). Each width w_i is a floating point number and it represents the width of all the possible abstract intervals associated to x_i. More precisely, given w_i and an integer index m, the interval uniquely associated to the variable x_i and the index m is

$$[m \times w_i, (m+1) \times w_i]$$

Then, since the width of each variable is fixed in the analysis, we can represent an interval for a floating point variable just using a single integer index.

Let us make a simple example. Consider a program with two variables, x_1, x_2. The widths associated to such variables are $w_1 = 3.0f, w_2 = 5.0f$. The vectors of the *GAS* will be vectors in two dimensions, $< \cdot, \cdot >$. The first element of each vector is an integer index representing the interval associated to x_1. The second element of each vector is an integer index representing the interval associated to x_2. So, the vector $v =< 1, 3 >$ indicates that the variable x_1 can assume values in the range $[3.0f, 6.0f]$ while the variable x_2 can assume values in the range $[15.0f, 20.0f]$. The indices can obviously also be negative (otherwise we would not be able to encode negative values). For example, the vector $w =< -2, -1 >$ is associated to the intervals $[-6.0f, -3.0f]$ for x_1 and $[-5.0f, 0.0f]$ for x_2. A *GAS* G containing only these two vectors ($G = \{v, w\}$) would mean that the possible values of x_1 are $\in [-6.0f, -3.0f] \vee [3.0f, 6.0f]$ and that the possible values of x_2 are $\in [-5.0f, 0.0f] \vee [15.0f, 20.0f]$. But we know more than that, because we also track relationships between variables, so we know that, when $x_2 \in [-5.0f, 0.0f]$ then $x_1 \in [-6.0f, -3.0f]$ (and viceversa), and the same for the other interval.

We can now formally define our abstract domain. Each abstract state is a set of vectors, where all the vectors are composed by n integer numbers (where n is the number of variables of the program, excluding constants which we are going to discuss in Section 4). The abstract domain definition (generic with respect to the number of program variables) is the following:

$$\mathcal{H} = \wp(\mathbb{Z}^n)$$

The lattice definition on this domain is immediate, since we exploit the usual abstract operators we find when abstract states are sets of elements: the partial

order is defined through set inclusion, the lub and glb are, respectively, set union and set intersection, while bottom and top are, respectively, the empty set and the set containing all possible n-dimensional vectors. Formally, the lattice definition is the following:

$$(\wp(\mathbb{Z}^n), \subseteq^{\mathcal{H}}, \cup^{\mathcal{H}}, \cap^{\mathcal{H}}, \bot^{\mathcal{H}}, \top^{\mathcal{H}})$$

where $\subseteq^{\mathcal{H}} = \subseteq, \cup^{\mathcal{H}} = \cup, \cap^{\mathcal{H}} = \cap, \bot^{\mathcal{H}} = \emptyset, \top^{\mathcal{H}} = \mathbb{Z}^n$.

As hinted before, for now we focus only on physics simulations, thus abstracting floating point variables through intervals. However, it is very easy to extend our framework to consider other types of variables (integer, boolean, etc.) or other kind of abstractions (signs, etc.), thus making our approach applicable to programs other than physics simulations. We will present an example of a different kind of application in Section 7.

The name of the $\mathcal{H}$ abstract domain (Hypercubes) comes from the geometric interpretation of a $\mathcal{GAS}$ vector (i.e., the combination of the various floating point intervals) in the n-dimensional space: each program variable corresponds to one dimension in the space and its interval corresponds to the hypercube coordinates in such dimension. The domain is called **Hypercubes** because a $\mathcal{GAS}$ is composed by a set of vectors and, as such, can be also seen as a set of hypercubes.

[5].

4 Abstract semantics

When creating an abstract domain, you also have to define the abstract versions of code operators which modify the variable values: in other words, you have to define the abstract semantics, i.e. the abstract version of the concrete semantics. For example, if you consider integer variables and you abstract them with the Sign domain, you also have to define the abstract version of arithmetic operators like $+$, -, and all other code operators which are commonly used on such variables. These abstract operations deal only with abstract values: for example, the abstract sum is an operation which takes two signs in input and returns another sign as output ($-\hat{+}- = -$, $-\hat{+}+ = \top$, etc).

In our domain, too, we have to define the abstract versions of interesting operators. In particular, since we deal with floating point variables abstracted with floating point intervals and physics simulations mainly execute arithmetic operations on them, we will have to define the abstract version of arithmetic operations with respect to floating point intervals. For example, we will have to define what is the result of the multiplication of two such intervals.

The Hypercubes domain, though, has also a peculiar feature, i.e. the existence of the global abstract state, $\mathcal{GAS}$. In fact, our analysis does not deal directly with an abstract value for each variable, but with a single $\mathcal{GAS}$, that is a set of vectors, each containing an abstract value for each variable. Then, how can

[5] **TODO GIULIA: aggiungere concretization e abstraction function**

we apply the usual abstract semantics (that is, abstract operations on floating point intervals) in our framework? The approach is the following:

- we consider, separately, each single vector of the current $\mathcal{GAS}$ (i.e., the $\mathcal{GAS}$ associated to the point of the program just before the considered statement)
- from each vector we "extract" the abstract values of the variables involved in the operation and we execute the abstract version of the operation with those abstract values, obtaining another abstract value (i.e., a floating point interval) as a result.
- we create a new vector, where the abstract value of the assigned variable (assuming that the operation is an assignment; we will discuss boolean conditions later) is the result of the abstract operation; all the other abstract values (including those of variables involved in the computation) are kept unmodified with respect to the original vector.

This approach, then, consists in creating a new vector for each vector of the current $\mathcal{GAS}$. The new $\mathcal{GAS}$ is the set containing all the newly created vectors. However, note that it does not always happen that a vector generates only one new vector. In fact, it could generate two vectors, or three, or more. Why can this happen? Because the floating point interval which results from the abstract operation could have a bigger width than the one fixed for the assigned variable, so we cannot represent such interval with a *single* integer index. To cover the entire interval, we must use more integer indices, and this leads to the creation of more than one new vector: we will create one new vector for each integer index resulting from the abstract operation. Remember that the other indices of the vector (that is, those associated to variables which do not get assigned) remain the same.

Let us see an example of the abstract semantics computation on a $\mathcal{GAS}$: consider a program `p` with three variables, $\mathtt{x}_1, \mathtt{x}_2, \mathtt{x}_3$. The widths associated to such variables are $\mathtt{w}_1 = 3.0f, \mathtt{w}_2 = 1.0f, \mathtt{w}_3 = 5.0f$. The considered statement is a sum: $\mathtt{x}_1 = \mathtt{x}_2 + \mathtt{x}_3$. The $\mathcal{GAS}$ G associated to the program `p` right before the execution of such statement is: $G = \{v_1 = [1, -1, -1], v_2 = [0, 2, 0]\}$, that is, the global state is composed by two vectors, v_1, v_2. We consider them one at a time:

- v_1: the intervals associated to $\mathtt{x}_2$ and $\mathtt{x}_3$ are, respectively, $[-1.0f, 0.0f]$ and $[-5.0f, 0.0f]$. The interval resulting from the sum of these two intervals is, intuitively [6], $[-6.0f, 0.0f]$. Since the width associated to $\mathtt{x}_1$, $\mathtt{w}_1$, is $3.0f$, a single index is not sufficient to cover the whole interval $[-6.0f, 0.0f]$: we need two indices, -2 and -1. Then, the vector v_1 generates two new vectors: $v_1' = [-2, -1, -1]$ and $v_1'' = [-1, -1, -1]$. Note that, in these two new vectors, the indices corresponding to x_2 and x_3 have not been modified with respect to their values in v_1.
- v_2: the intervals associated to $\mathtt{x}_2$ and $\mathtt{x}_3$ are, respectively, $[2.0f, 3.0f]$ and $[0.0f, 5.0f]$. The interval resulting from the sum of these two intervals is $[2.0f, 8.0f]$. This interval needs three indices to be represented with the width

[6] we will formally define the abstract sum operation shortly

of $\mathtt{x}_1$: 0, 1 and 2. Then, v_2 generates three new vectors: $v_2' = [0, 2, 0]$, $v_2'' = [1, 2, 0]$ and $v_2''' = [2, 2, 0]$.

The new $\mathcal{GAS}$ G' resulting from the execution of the sum statement is then $G' = \{v_1', v_1'', v_2', v_2'', v_2'''\}$.

Note that, to execute an abstract operation, we have to consider each vector in the $\mathcal{GAS}$ and the abstract values of all variables involved in the operation. Anyway, when we focus on one vector, we execute the abstract operation considering *only* variable values coming from the *same* vector. We do not check the values of the variables involved in the operation in all other vectors, because we want to maintain the relationships between variables. For this reason, the computation of the abstract semantics of a code statement has a linear complexity with respect to the number of vectors in the $\mathcal{GAS}$, but it is not quadratic or, worse, exponential, because it does not depend in any way on the number of variables involved in the operation.

4.1 Dealing with constants

We said that the $\mathcal{GAS}$ vectors have an element for each variable of the program. However, we exclude from the vectors the constants of the program. We define a constant as a variable which gets assigned only once, or a numeric value which appears in some statement (without being assigned to a variable).

Before starting the analysis we have to determine the size of the vectors. To do this, we must find all the variables of the code which are not constants. To avoid scanning the entire code before starting the analysis, we require the program to initialize all non-constant variables at the beginning of the program. Constant variables can be assigned later in the code, since they do not influence the size of the vectors. Note that physics simulations (which is the main application field we consider) are generally made up by an initialization of all variables, followed by a `while` loop. In such case, the program respects our condition, and it is immediate to find out which variables are not constants and, therefore, the size of the vectors.

What do we do with constants? When we encounter a program statement which contains a constant (both in the form of an assign-once variable or in the form of an explicit number), the abstract semantics considers such constant as an interval of zero width: the extremes of the interval are the same and they are equal to the value of the constant. In this way we get to deal uniformly with all entities involved in the abstract semantics computation.

For example, in a physics simulation, a variable which is usually a constant is *dt*, i.e. the amount of time which passes between two iterations of the while loop. If the initialization of such variable was `dt = 0.006`, then this constant-variable would be associated to the interval $[0.006, 0.006]$.

4.2 Boolean conditions semantics

We now present the abstract semantics of boolean conditions. First of all, by boolean condition we mean a statement which contains a comparison operator like $>, <, \leq, \geq, \neq$. The procedure to deal with such a statement is as follows:

- we separately execute the abstract semantics on the two parts of the statement, the one on the left of the comparison operator and the one on the right.
- if a constant appears in one of the two parts of the statement (a single part could be made up by *only* a constant), we deal with it as explained in 4.1.
- we obtain two floating-point intervals to compare. Let us call them $i_1 = [a, b]$ and $i_2 = [c, d]$. The abstract comparison between them depends on the specific comparison operator present in the statement:
  - $i_1 \neq i_2$ returns true if $b < c \vee a > d$, top otherwise (by top we mean the top element of the boolean abstract domain).
  - $i_1 < i_2$ returns true if $b < c$, false if $a > d$, top otherwise.
  - $i_1 > i_2$ returns true if $a > d$, false if $b < c$, top otherwise.
  - $i_1 \leq i_2$ returns true if $b \leq c$, false if $a \geq d$, top otherwise.
  - $i_1 \geq i_2$ returns true if $a \geq d$, false if $b \leq c$, top otherwise.

The result of the boolean condition is then an abstract value of the boolean domain, obtained by executing the abstract semantics on the two parts of the statement and then comparing the two resulting intervals.

4.3 If-then-else semantics

Given the particular definition of our domain, we use a different (and more precise) abstract semantics for the conditional branches (that is, `if − then − else` statements) with the respect to the usual one. Remember that a $\mathcal{GAS}$ is a set of vectors, and each vector represents an admissible portion of the variable space (at a certain point of the program). Our over-approximation of the state of a program (at a specific point) is the union of all these portions of space.

To deal with branches, we partition the $\mathcal{GAS}$ with respect to the branching condition. In particular, for each vector of the $\mathcal{GAS}$ we extract the abstract value of all the variables involved in the computation of the branching condition and we execute the abstract semantics of the condition with these values (see Section 4.2). We assign each vector to a specific partition, based on the result of the condition computation. The partitions we obtain are three: the vectors for which the condition evaluates to true (partition $\mathtt{p_t}$), the ones for which the condition evaluates to false (partition $\mathtt{p_f}$), and the ones for which we do not have a definitive answer (i.e., the result of the evaluation is the top value of the boolean domain), partition $\mathtt{p_T}$. A vector of the $\mathcal{GAS}$ can belong to only one of these three partitions. Calling G the $\mathcal{GAS}$, this means that: $G = \mathtt{p_t} \cup \mathtt{p_f} \cup \mathtt{p_T}$. Also, the partitions have no intersections between each others.

Once obtained these three partitions, we can execute selectively the abstract semantics of the two branches (`then` and `else`). In particular:

– the vectors of partition $\mathtt{p_t}$ represent the portions of the variable space for which the branching condition is surely true. Then, we can execute on such vectors *only* the statements from the `then` branch. This results in a set of new vectors, which we call $\mathtt{p'_t}$.
– the vectors of partition $\mathtt{p_f}$ represent the portions of the variable space for which the branching condition is surely false. Then, we can execute on such vectors *only* the statements from the `else` branch. This results in a set of new vectors, which we call $\mathtt{p'_f}$.
– the vectors of partition $\mathtt{p_T}$ represent the portions of the variable space for which the branching condition could be both true or false. Then, we have to execute on such vectors both the statements from the `then` branch and those from the `else` branch, separately. This results in two sets of new vectors, which we call $\mathtt{p'_T}$ (i.e., the result of the execution of the `then` branch on the partition $\mathtt{p_T}$) and $\mathtt{p''_T}$ (i.e., the result of the execution of the `else` branch on the partition $\mathtt{p_T}$).

Then, we compute the least upper bound of all vectors sets obtained with this schema. Remember that the lub of our domain is set union. The result of the abstract semantics of the `if` − `then` − `else` when starting from the $\mathcal{GAS}$ G is then:

$$\mathtt{p'_t} \cup \mathtt{p'_f} \cup \mathtt{p'_T} \cup \mathtt{p''_T}$$

4.4 Arithmetic operations semantics

In this Section we define the abstract semantics of the most used arithmetic operators. In particular, we consider sum (+), product (×), subtraction (−), division (/), change of sign ($-(\cdot)$) and modulus (||). These operators should suffice for most physics simulations (for example, our case study requires only sum, product and change of sign). Anyway, our framework can be easily extended to support other operations.

semantica delle operazioni astratte (somma, prodotto, etc). dire che se usiamo diverse rappresentazioni (per esempio i segni) dobbiamo specificare anche le nuove versioni delle operazioni astratte che coinvolgono quel tipo di dato (per esempio, + per - risulta -, 0 a cui sommi uno risulta +, etc).

5 Implementation

In this Section we discuss how we implemented our technique in a functional framework. We think this observation is important because we argue that a different implementation approach would have required more effort on the part of the developer and would not have permitted us to obtain experimental results so quickly. bla bla

6 Assessment

In this Section we present the results of our approach when analysing the case study of Section 2. The scenario offered in our case study is challenging because ... bla bla

NB: present also the results of the case study using the simple Interval domain. Tweak the various parameters (initial values, widths, dt, etc) to make the Interval domain fail, while the Hypercubes verifies the property.

7 Other applications

In this Section we show how our approach is generic with respect to the field of application. In fact, our technique can be effectively used also outside the field of physics simulations. Here we present a program bla bla ... (programma con x e y che hanno sempre segno opposto) and we show how we can prove an interesting property. Other existing approaches (signs, intervals, pohlyedra, and so on) are not able to prove the same property.

8 Related work

In this Section we discuss the related work.

9 Conclusions

In this section we discuss the the overall structure of our work, its original intentions, and the results of its assessment.

10 Future work

In this Section we discuss possible extensions of this work in order to improve its performance and precision. There are a lot of possibilities, but the ones we deem more urgent and interesting are the following:

- multi-level grid for each variable
- arbitrary sub-volumes for each hypercubes (first of all, offsets within the ranges)
- a smart widening operator which understands the "direction" of the program (i.e., its derivative) and is able to deal with properties associated to a generic amount of time t (i.e., "it *never/always* happens that ... ").

11 References