The Domain of Parametric Hypercubes for Static Analysis of Computer Games Software

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

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

Abstract. Computer Games Software deeply relies on physics simulations, which are particularly demanding to analyze because they manipulate a large amount of interleaving floating point variables. Therefore, this application domain is an interesting workbench to stress the trade-off between accuracy and efficiency of abstract domains for static analysis. In this paper, we introduce Parametric Hypercubes, a novel disjunctive non-relational abstract domain. Its main features are: (i) it combines the low computational cost of operations on (selected) multidimensional intervals with the accuracy provided by lifting to a power-set disjunctive domain, (ii) the compact representation of its elements allows to limit the space complexity of the analysis, and (iii) the parametric nature of the domain provides a way to tune the accuracy/efficiency of the analysis by just setting the widths of the hypercubes sides.
The first experimental results on a representative Computer Games case study outline both the efficiency and the precision of the proposal.

1 Introduction

Computer Games Software is a fast growing industry, with more than 200 million units sold every year, and annual revenue of more than 10 billion dollars. According to the Entertainment Software Association (ESA), more than 25% of the software played concerns sport, action, and strategy games, where physics simulations are the core of the product, and compile-time verification of behavioural properties is particularly challenging for developers.

The difficulty arises because, usually, these programs feature (i) a `while` loop which goes on endlessly, (ii) a complex state made up by multiple real-valued variables, and (iii) strong dependencies among variables. Most of the times, a simulation consists in the initialization of the state (i.e., the variables which compose the simulated world) followed by an infinite `while` loop which computes the numerical integration over time (i.e., the inductive step of the simulation). Such loop is executed until the game is stopped. In addition, the variables of

a physics simulation are real-valued, because they represent continuous values that map directly to physical aspects of the real world, like positions, velocities (speed plus direction), and accelerations. Finally, the variables of a simulation are strongly inter-related, because the simulation often 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 object position. 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.

Interesting properties on physical programs are, for example, the insurance that a rocket reaches a stable orbit, or that a bouncing ball arrives at a certain destination. To prove such properties statically, we need to precisely track relationships between variables. However, traditional approaches to static analysis are not best suited to deal with these kind of properties. On the one hand, non-relational domains are usually too approximate. On the other hand, the computational cost of sophisticated relational domains like Polyhedra [10] or Parallelotopes [3] is too high, and their practical use in this context becomes unfeasible.

In this paper, we introduce Parametric Hypercubes, a novel disjunctive non-relational abstract domain. Its main features are: (i) it combines the low computational cost of operations on (selected) multidimensional intervals with the accuracy provided by lifting to a power-set domain, (ii) the compact representation of its elements allows to limit the space complexity of the analysis, and (iii) the parametric nature of the domain provides a way to tune the trade-off between accuracy and efficiency of the analysis by just setting the widths of the hypercubes sides. The domain can be seen as the combination of a suite of well-known techniques for numerical abstract domain design, like disjunctive powerset, and conditional partitioning. The most interesting points of our work are: (i) the approach: the design of the domain has as starting point the features of the application domains, (ii) the self-adaptive parameterization: a recursive algorithm is applied to refine the initial set of parameters in order to improve the accuracy of the analysis without sacrificing the performance, and (iii) the novel notion of "offset" that allows to narrow the lack of precision due to the fixed width of intervals. The analysis has been implemented, and it shows promising results in terms both of efficiency and precision when applied to a representative case study of Computer Games Software.

The rest of the paper is structured as follows. Section 2 presents the language syntax supported by our analysis and Section 3 introduces the case study which we use to experiment with our approach. Sections 4 and 5 formally define the abstract domain and semantics, respectively. Section 6 contains the experimental results of our analysis applied to the case study of Section 3. Section 7 presents the related work and Section 8 concludes.

2 Language syntax

Let $\mathcal{V}$ be a finite set of variables, and $\mathcal{I}$ the set of all real-valued intervals. Figure 1 defines the language. We focus on programs dealing with mathematical computations over real-valued variables. Therefore, we consider expressions built through the most common mathematical operators (sum, subtraction, multiplication, and division). An arithmetic expression can be a constant value ($c \in \mathbb{R}$), a non-deterministic value in an interval ($I \in \mathcal{I}$), or a variable ($V \in \mathcal{V}$). We also consider boolean conditions built through the comparison of two arithmetic expressions. Boolean conditions can be combined as usual with logical operators (and, or, not). As for statements, we support the assignment of an expression to a variable, `if` $-$ `then` $-$ `else`, `while` loops, and concatenation. Even though this syntax is simple and limited, many physical simulations can be built through it [6], since their complexity lies mostly in their logic and not in the used constructs.

$$V \in \mathcal{V}, I \in \mathcal{I}, c \in \mathbb{R}$$
$$E := c|I|V|E \ < aop > \ E \text{ where } < aop > \in \{+, -, \times, \div\}$$
$$B := E < bop > E|B \ and \ B|not \ B|B \ or \ B \text{ where } < bop > \in \{\geq, >, \leq, <, \neq\}$$
$$P := V = E|\texttt{if}(B) \ \texttt{then} \ P \ \texttt{else} \ P|\texttt{while}(B) \ P|P; P$$

Fig. 1. Syntax

3 The case study of bouncing balls

Consider the program in Figure 2. It generates a bouncing ball that starts at the left side of the screen (even though the exact initial position is not fixed), and it has a random initial velocity. The horizontal direction of the ball is always towards the right of the screen, since `vx` ≥ 0.Whenever the ball reaches the bottom of the screen, it bounces (i.e., its vertical velocity is inverted). When the ball reaches the right border of the screen, it disappears. We want to verify that T seconds after the generation of the ball, such ball has already exited from the screen (we call this *Property 1*).

The structure of this program respects the generic structure of a physics simulation, as explained in Section 1. The meanings of the variables are as fol-

```
let px = rand(0.0, 10.0), py = rand(0.0, 50.0)
let vx = rand(0.0, 60.0), vy = rand(−30.0, −25.0)
let dt = 0.05, g = −9.8, k = 0.8

while (true) do
    if ( py >= 0.0 ) then
      (px, py) = (px + vx ∗ dt, py + vy ∗ dt)
      (vx, vy) = (vx, vy + g ∗ dt)
    else
      (px, py) = (px + vx ∗ dt, 0.0)
      (vx, vy) = (vx, −vy) ∗ k
```

Fig. 2. Case study: bouncing-ball code

lows. (px, py) represents the current position of the ball in the screen, and its initial values are generated randomly. (vx, vy) represents the current velocity of the ball, and its initial values are generated randomly as well. dt represents the time interval between iterations of the loop. This value is constant and known at compile time (dt = 1/20 = 0.05 considering a simulation running at 20 frames per second). g represents the force of gravity (−9.8). k represents how much the impact with the ground decreases the velocity of the ball.

The while loop updates the ball position and velocity. To simulate the bouncing, we update the horizontal position according to the rule of uniform linear motion, while we force the vertical position to zero when the ball touches the ground and we invert the vertical velocity. In addition, we decrease both the horizontal and vertical velocity through the constant factor k, to consider the force which is lost in the impact with the ground.

```
let balls = Set.empty
let dt = 0.05, creationInterval = 3.0, timeFromLastCreation = 0.0
while (true) do
        foreach ball in balls
                updateBall(ball)
        if (timeFromLastCreation >= creationInterval)
                generateNewBall()
                timeFromLastCreation = 0.0
        else
                timeFromLastCreation += dt
```

Fig. 3. Bouncing ball generation

Verifying Property 1 on this program has a significant practical interest, since it is a basic physics simulation which can be used in many contexts [11]. For instance, consider the program in Figure 3, where updateBall(b) moves the ball b (through the body of the while loop of Figure 2) and generateNewBall() creates a new ball (with the values of the initialization of Figure 2). It discreetly generates bouncing balls on the screen. The interval between the creation of two balls (creationInterval) is constant and known at compile time.

Proving *Property 1* on the program in Figure 2 means that a single ball will have exited the screen after T seconds. In addition, in the program of Figure 3, we generate one ball each creationInterval seconds. This means that, having verified *Property 1*, we can guarantee that *a maximum of* $\lceil \frac{T}{\texttt{creationInterval}} \rceil$ *balls will be on the screen at the same time*. Such information may be useful for performance reasons (crucial in a game), since each ball requires computations for its rendering and updating.

Non-disjunctive or non-relational static analyses are not properly suited to verify *Property 1*. Consider for example the Interval domain where every variable of the program is associated to a single interval. After a few iterations, when the vertical position possibly goes to zero, the analysis is not able to distinguish which branch of the if − then − else to take anymore. In this case, the lub operator makes the vertical velocity interval quite wide, since it will contain both positive and negative values. After that, the precision gets completely lost, since the velocity variable affects the position and vice-versa. On the other hand,

the accuracy that would be ensured by using existing disjunctive domains has a computational cost that makes this approach unfeasible for practical use.

4 The Parametric Hypercubes domain

Intuitively, an abstract state of the Parametric Hypercubes domain ($\mathcal{H}$) tracks disjunctive information relying on floating-point intervals of fixed width. A state of $\mathcal{H}$ is made by a set of hypercubes of dimension $|\mathsf{Vars}|$. Each hypercube has $|\mathsf{Vars}|$ sides, one for each variable, and each side contains an abstract non-relational value for the corresponding variable. Each hypercube represents a set of admissible combinations of values for all variables.

The name Hypercubes comes from the geometric interpretation of the elements of $\mathcal{H}$. The concrete state of a program with variables in Vars is an environment in $\mathsf{Vars} \to \mathbb{R}$. This can be isomorphically represented by a tuple of values where each item of the tuple represents a program variable. Seen in this way, the concrete state corresponds, geometrically, to *a point* in the $|\mathsf{Vars}|$-dimensional space. The hypercubes of our domain $\mathcal{H}$ are *volumes* in the same $|\mathsf{Vars}|$-dimensional space.

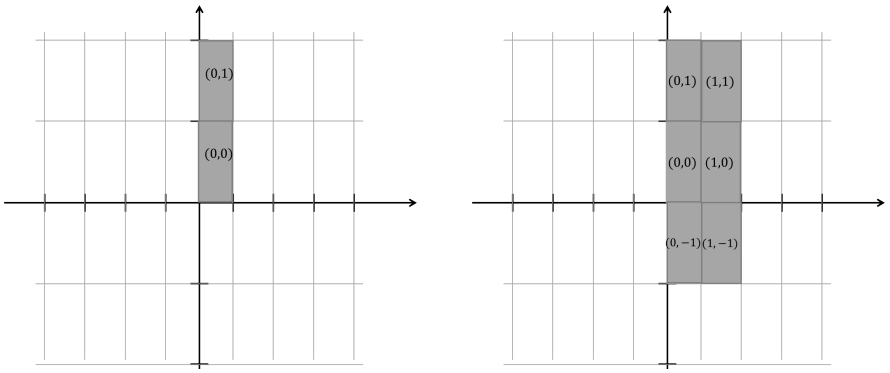
4.1 Lattice structure

An abstract state of $\mathcal{H}$ tracks a *set* of hypercubes, and each hypercube is represented by a tuple of abstract values. The dimension of these tuples is equal to the number of program variables. We abstract floating-point variables through intervals of real values. A set of hypercubes allows us to track disjunctive information, and this is useful when the values of a variable are clustered in different ranges. The performance of this domain, though, becomes a crucial point, because the number of possible hypercubes in the space is potentially exponential with respect to the number of partitions along each spatial axis.

First of all, the complexity is lightened by the use of a *fixed* width for each variable, by partitioning the possible intervals, and by the efficiency of set operators on tuples. Then, another performance booster is the use of a smart representation for intervals: in order to store the specific interval range we just use a single integer representing it. This is possible because each variable x_i is associated to an interval width (specific only for that variable), which we call w_i and which is a parameter of the analysis. Each width w_i represents the width of all the possible abstract intervals associated to x_i. More precisely, given a width w_i and an integer index m, the interval uniquely associated to the variable x_i is $[m \times w_i..(m+1) \times w_i]$. Notice that the smaller the width associated to a variable, the more granular and precise the analysis on that variable (and the heavier computationally the analysis). In Section 5 we will show how to compute and adjust automatically the widths.

Example: Consider the case study of Section 3 and in particular the two variables `px` and `py`. Suppose that the widths associated to such variables are $w_1 = 10.0, w_2 = 25.0$. The hypercubes in this case are 2D-rectangles that can be represented on the Cartesian plane. Each side of a hypercube is identified

by an integer index, and a 2D hypercube is then uniquely identified by a pair of integers. For instance, the hypercube $h_1 = (0, 1)$ represents $\texttt{px} \in [0.0..10.0]$ and $\texttt{py} \in [25.0..50.0]$, while the hypercube $h_2 = (0, 0)$ associates $\texttt{px}$ to $[0.0..10.0]$ and $\texttt{py}$ to $[0.0..25.0]$. Figure 4a depicts the two hypercubes associated to the initialization of the case study (i.e., h_1 and h_2). Instead, Figure 4b depicts the six hypercubes obtained after executing the first iteration of the `while` loop.



(a) The abstract state after the initialization of the variables $\texttt{px}, \texttt{py}$, when their widths are, respectively, 10.0 and 25.0

(b) The abstract state of $\texttt{px}, \texttt{py}$ after the first iteration of the loop (widths are, respectively, 10.0 and 25.0)

Fig. 4. Cartesian plans

We now formalize our abstract domain. Each abstract state is a set of hypercubes, where each hypercube is composed by $|\mathsf{Vars}|$ integer numbers. The abstract domain is then defined by $\mathcal{H} = \wp(\mathbb{Z}^n)$ where $n = |\mathsf{Vars}|$. The definition of lattice operators relies on set operators. Formally, $\langle \wp(\mathbb{Z}^n), \subseteq, \cup, \cap, \emptyset, \mathbb{Z}^n \rangle$.

4.2 Concretization function

We denote by $\mathcal{A}$ the non-relational abstract domain on which our analysis is parameterized, and by n the number of variables of the program. Let $\sigma \in \mathbb{R}^n$ be a tuple and $\sigma_i \in \mathbb{R}$ be the i-th element of such tuple. Also, let $\gamma_{\mathcal{A}} : \mathcal{A} \to \wp(\mathbb{R})$ be the concretization function of abstract values of the non-relational abstract domain $\mathcal{A}$, and $getAbsValue_v : \mathbb{N} \to \mathcal{A}$ be the function that, given an integer index, returns the abstract value (in the domain $\mathcal{A}$) which corresponds to that index inside the tuple v. Then, the function $\gamma_{\mathsf{Val}} : \wp(\mathcal{A}^n) \to \wp(\mathbb{R}^n)$ concretizes a set of hypercubes to a set of vectors of n floating point values. Formally, $\gamma_{\mathsf{Val}}(\mathsf{V}) = \{\sigma : \exists v \in \mathsf{V} : \forall i \in [1..n] : \sigma_i \in \gamma_{\mathcal{A}}(getAbsValue_v(i))\}$ where $\mathsf{V} \in \wp(\mathcal{A}^n)$ is a set of hypercubes. Finally, based on γ_{Val}, we can define the function $\gamma_{\mathcal{H}}$, which maps a subset V of $\wp(\mathcal{A}^n)$ into an environment. The function $\gamma_{\mathcal{H}} : \wp(\mathcal{A}^n) \to \wp(\mathsf{Vars} \to \mathbb{R})$ concretizes the hypercubes domain. Formally, $\gamma_{\mathcal{H}}(\mathsf{V}) = \{[\mathtt{x} \mapsto \sigma_{varIndex(\mathtt{x})} : \mathtt{x} \in \mathsf{Vars}] : \sigma \in \gamma_{\mathsf{Val}}(\mathsf{V})\}$. The function $\gamma_{\mathcal{H}}$ maps the vectors returned by γ_{Val} into concrete environments relying on the function $varIndex : \mathsf{Vars} \to \mathbb{N}$. The latter, given a variable, returns its index in the tuples which compose the elements of $\mathcal{H}$.

4.3 Convergence of the analysis

The number of hypercubes in an abstract state may increase indefinitely. In order to make the analysis convergent, we fix for each variable of the program a maximum integer index n_i such that n_i represents the interval $[n_i \times w_i .. +\infty]$. The same happens symmetrically for negative values. In this way, the set of indices of a given variable is finite, the resulting domain has finite height, and the analysis is convergent.

This approach may seem too rough since we establish the bounds of intervals before running the analysis. However, when analysing physics simulations we can use the initialization of variables and the property to verify in order to establish convenient bounds for the intervals. For instance, in the case study presented in Section 3 we are interested in checking if a ball stays in the screen, that is, if `px` is greater than zero and less than a given value `w` representing the width of the screen. Since we are only interested in proving that, once a ball has exited the screen, it does not come back, we can abstract together all the values that are greater than `w`.

4.4 Offsets

A loss of precision may occur due to the fact that hypercubes proliferate too much, even using small widths. Consider, for example, the statement `x = x + 0.01` (which is repeated at each iteration of the `while` loop) with 1.0 as the width associated to `x`. If [0.0..1.0] was the initial interval associated to `x`, the sequence of abstract states would be: {[0.0..1.0]}, {[0.0..1.0], [1.0..2.0]}, {[0.0..1.0], [1.0..2.0], [2.0..3.0]} and so on. At each iteration we would add one interval.

In order to overcome these situations, we further improve the definition of our domain: in each hypercube, each variable v_i (associated to width w_i) is related to (other than an integer index i representing the fixed-width interval $[i \times w_i .. (i+1) \times w_i]$) a specific offset (o_m, o_M) *inside* such interval. In this way, we use a sub-interval (of arbitrary width) inside the fixed-interval width, thereby restricting the possible values that the variable can assume. Both o_m and o_M must be smaller than w_i, greater than or equal to 0 and $o_m \leq o_M$. Then, if i and (o_m, o_M) are associated to v_i, this means that the possible values of v_i belong to the interval $[(i \times w_i) + o_m .. (i \times w_i) + o_M]$.

An element of our abstract domain is then stored as a map from hypercubes to tuples of offsets. In this way, we can keep the original definition of a hypercube as a tuple of integers, but we also map each hypercube to a tuple of offsets (one for each variable). Now an abstract state is defined by $M : \mathbb{Z}^{|Vars|} \to (\mathbb{R} \times \mathbb{R})^{|Vars|}$, i.e., a map where the domain is the set of hypercubes, and the codomain is the set of tuples of offsets.

The least upper bound between two abstract states ($M = M_1 \sqcup M_2$) is then defined by $dom(M) = dom(M_1) \cup dom(M_2)$, and

$$\forall h \in dom(M) : M(h) = \begin{cases} M_1(h) & \text{if } h \in dom(M_1) \wedge h \notin dom(M_2) \\ M_2(h) & \text{if } h \in dom(M_2) \wedge h \notin dom(M_1) \\ merge(M_1(h), M_2(h)) & \text{otherwise} \end{cases}$$

where $merge(o_1, o_2)$ creates a new tuple of offsets by merging the two tuples of offsets in input: for each pair of corresponding offsets (for example (m_1, M_1) and (m_2, M_2)), the new offset is the widest combination possible (i.e., $(\min(m_1, m_2)$ and $\max(M_1, M_2))$). Note that this definition corresponds to the pointwise application of the least upper bound operator over intervals. The widening operator is extended in the same way: it applies the standard widening operators over intervals pointwisely to the elements of the vector representing the offsets.

5 Abstract semantics

For the most part, the abstract semantics applies existing semantic operators of boxed Intervals [9]. In this section, we sketch how these operators are used to define the semantics on $\mathcal{H}$.

First of all, $\mathbb{I}$ defines the semantics of arithmetic expressions on a single hypercube by applying the well-known arithmetic operators on intervals.

We use the semantics $\mathbb{I}$ to define the abstract semantics $\mathbb{B}$ of Boolean comparisons. Given a hypercube and a Boolean comparison $E_1 < bop > E_2$ where $< bop > \in \{\geq, >, \leq, <, \neq\}$, $\mathbb{B}$ returns an *abstract value of the boolean domain* (namely, *true*, *false*, or $\top$) comparing the intervals obtained from E_1 and E_2 through $\mathbb{I}$. Therefore, given a Boolean condition and a set of hypercubes, we partition this set into the hypercubes for which (i) the condition surely holds, (ii) the condition surely does not hold, and (iii) the condition may or may not hold. In this way, we can discard all the hypercubes for which a given Boolean condition surely holds or does not hold. The semantics of the logical operators *not*, *and*, *or* is defined in the standard way.

$\mathbb{I}$ is used to define the semantics $\mathbb{S}$ of variable assignment as well. The standard semantics of $\texttt{x} = \texttt{exp}$ is to (i) obtain the interval representing the right part ($\mathbb{I}[\![\texttt{x} = \texttt{exp}, \sigma]\!] = [m..M]$), and (ii) assign it in the current state. This approach does not necessarily produce a single hypercube, since the interval to assign could have a greater width than the fixed width of the assigned variable (for example, the interval $[0..6]$ when $w = 5$). It could also happen that the resulting interval width is smaller than the fixed width, but the interval spans over more than one hypercube side, due to the fixed space partitioning (for example, the interval $[3..6]$ when $w = 5$, because the space is partitioned in $[0..5], [5..10]$, etc.). In these cases, we build up several hypercubes that cover the interval $[m..M]$. This can be formalized by $assign(h, V_i, [a..b]) = \{h[i \mapsto m] : [m \times w_i..(m+1) \times w_i] \cap [a..b] \neq \emptyset\}$, where h is a hypercube, V_i is the assigned variable, and $[a..b]$ is the interval we are assigning (which depends on the hypercube h, since we use its variables values to compute the result of the expression). We repeat this process for each hypercube h in the abstract state by using it as input for the computation of *assign*. In this way, we are able to over-approximate the assignment while also keeping the fixed widths of the intervals, which are very important for performance issues.

Offsets Offsets allow us to recover some precision when computing the abstract semantics of assignment. In particular, as the expression semantics $\mathbb{I}$ returns intervals of arbitrary widths, we can use such exact result to update the offsets

of the abstract state. Formally, the semantics of the assignment is defined as follows:

$$assign(h, V_i, [a..b]) = \{h[i \mapsto (m, o_m, o_M)] : [m \times w_i..(m+1) \times w_i] \cap [a..b] \neq \emptyset\}$$

where h is a hypercube, V_i is the assigned variable, $[a..b]$ is the interval we are assigning and o_m, o_M are computed as:

$$o_m = \begin{cases} 0 & \text{if } a \leq (m \times w_i) \\ a - (m \times w_i) & \text{otherwise} \end{cases} \wedge o_M = \begin{cases} w_i & \text{if } b \geq ((m+1) \times w_i) \\ b - (m \times w_i) & \text{otherwise} \end{cases}$$

Note that, when we extract from a hypercube the interval associated to a variable, we use the interval delimited by the offsets, so that abstract operations can be much more precise.

Consider the evaluation of statement `x = x + 0.01` inside a while loop with 1.0 as width of `x` and [0..1] as initial value of `x`. After the first iteration, the abstract semantics computes [0.0..1.0] and [1.0..2.0] with offsets [0.01..1.0] and [1.0..1.01], respectively. In this way, at the following iteration we would obtain again the same two intervals with the offsets changed to [0.02..1.0] and [1.0..1.02]. This results is strictly more precise than the one obtained without offsets, and it is an essential feature of our abstract domain. For instance, in the case study of Figure 2 offsets will allow us to discover if a bouncing ball exits the screen after N iterations of the `while` loop.

5.1 Initialization of the analysis

Before starting the analysis we have to determine the number of sides each hypercube will have. To do this, we must find all the variables ($Vars$) of the program which are not constants (i.e., assigned only once at the beginning of the program). We require the program to initialize all the variables at the beginning of the program. The initialization of the analysis is made in two steps. First, for each initialized variable, we compute its abstraction in the non-relational domain chosen to represent the single variables. The resulting set of abstract values could contain more than one element. Let us call $\alpha(V)$ the set of abstract values associated to the initialization of the variable $V \in Vars$. Then we compute the Cartesian product of all sets of abstracted values (one for each variable). The resulting set of tuples (where each tuple has the same cardinality as $Vars$) is the initial set of hypercubes of the analysis. Formally, $\mathcal{H} = \bigtimes_{V \in Vars} \alpha(V)$.

Consider the code of our case study in Figure 2. First of all, we must identify the variables which are not constants: dt, g, k are assigned only during the initialization, so we do not include them in $Vars$. The set of not-constant variables is then $Vars = \{V_1 = px, V_2 = py, V_3 = vx, V_4 = vy\}$, and so $|Vars| = 4$.

5.2 Tracking the origins

During the analysis of a program we also track, for each hypercube of the current abstract state, the initial hypercubes (*origins*) from which it is derived. To store such information, we proceed as follows. Let H_i be the set of hypercubes obtained

for the i-th statement of the program. The data structure of a hypercube h contains also an additional set of hypercubes, h^{or}, which are its origins and are always a subset of the initial set of hypercubes, i.e., $\forall h : h^{or} \subseteq H_0$. At the first iteration, each hypercube contains only itself in its origins set: $\forall h \in H_0 : h^{or} = \{h\}$. When we execute a statement of the program, each hypercube produces some new hypercubes: at this stage, the origins set is simply propagated. For example, if h generates h_1, h_2, then $h_1^{or} = h_2^{or} = h^{or}$. When merging all the newly produced hypercubes in a single set (the abstract state associated to the point of the program just after the executed statement), we also merge through set union the sets of origins of any repeated hypercube. For example, consider $H_i = \{h_a, h_b\}$ and let h_1, h_2 be the hypercubes produced by h_a executing statement i-th and h_2, h_3 be those produced by h_b. Then, $H_{i+1} = \{h_1, h_2, h_3\}$ and $h_1^{or} = h_a^{or}$, $h_2^{or} = h_a^{or} \cup h_b^{or}$ and $h_3^{or} = h_b^{or}$.

5.3 Width choice

The choice of the interval widths influences both the precision and efficiency of the analysis. On the one hand, if we use smaller widths we certainly obtain more precision, but the analysis risks to be too slow. On the other hand, with bigger widths the analysis will be surely faster, but we could not be able to verify the desired property. To deal with this trade-off, we implemented a recursive algorithm which adjusts the widths automatically. We start with wide intervals (i.e., coarse precision, but fast results) and we run the analysis for the first time. At the end of the analysis, we check, for each hypercube of the *final* set, if it verifies the desired property. We then associate to each *origin* (i.e., initial hypercube) its final result by merging the results of its derived final hypercubes (we know this relationship because of the origins set stored in each hypercube): some origins will certainly verify the property (i.e., they produce only final hypercubes which satisfy the property), some will not, and some will not be able to give us a definite answer (because they produce both hypercubes which verify the property and hypercubes which do not verify it). We partition the starting hypercubes set with respect to this criterion (obtaining, respectively, the *yes* set, the *no* set and the *maybe* set), and then we run the analysis again with halved widths, but *only* on the origins which did not give a definite answer (the *maybe* set). This step is only performed until we reach a specific threshold, i.e., the *minimum width* allowed for the analysis. The smaller this threshold is, the more precise (but slower) the analysis becomes.

The analysis is then able to tell us which initial values of the variables bring us to verify the property (the union of all the *yes* sets encountered during the recursive algorithm) and which do not. Thanks to these results, the user can modify the initial values of the program, and run the analysis again, until the answer is that the property is verified *for all initial values*. In our case study, for example, we can adjust the possible initial positions and velocities until we are sure that the ball will exit the screen in a certain time frame.

The formalization of this recursive algorithm is presented in Algorithm 1.

Algorithm 1 The width adjusting recursive algorithm

function ANALYSIS($currWidth, minWidth, startingHypercubes$)
  return $(yes \cup yes', no \cup no', maybe')$
  where
  $(yes, no, maybe) = hypercubesAnalysis(currWidth, startingHypercubes)$
  if $currWidth/2.0 \geq minWidth$ **then**
    $(yes', no', maybe') = Analysis(currWidth/2.0, minWidth, maybe)$
  else
    $(yes', no', maybe') = (Set.empty, Set.empty, maybe)$
  end if
end function

6 Experimental results

In this Section we present some experimental results on the case study presented in Section 3. We want to check if *Property 1* is verified on the program of Figure 2 and, in particular, we want to know which subset of starting values brings to verify it. We implemented our analysis in the F# language with Visual Studio 2012. We ran the analysis on an Intel Core i5 CPU 1.60 GHz with 4 GB of RAM, running Windows 8 and the F# runtime 4.0 under .NET 4.0.

We set the initial widths associated to all variables to 100.0 and the minimum width allowed to 5.0. As for *Property 1*, we set $T = 5$, i.e., we want to verify if the ball is surely out of the screen within 5 seconds from its generation. Since $dt = 0.05$, a simulation during 5 seconds corresponds to $5/0.05 = 100$ iterations of the `while` loop. To verify this property, we apply trace partitioning [18] to track one abstract state per loop iteration until the 100-th iteration (we do not need to track precise information after the 100th iteration). The position which corresponds to the exiting from the screen is 100.0: if after 100 iterations the position `px` is surely greater than 100.0, then *Property 1* is verified. The whole of these values (starting variables values and widths, minimum width allowed, number of iterations, position to reach) make up our *standard workbench data*. We will experiment to study how efficiency and precision change when modifying some parameters of the analysis.

For each test, the analysis returns three sets of starting hypercubes: the initial values of the variables which satisfy the property (*yes* set), which surely do not satisfy the property (*no* set), and which may or not satisfy the property (*maybe* set). To make the results more immediate and clearer, we computed for each *yes* and *no* set the corresponding volume covered in the space by their hypercubes. We also consider the *total* volume of the variable space, i.e., the volume covered by all possible values with which the program variables are initialized. In the case of the standard workbench data, the *total* volume is $10.0 \times 50.0 \times 60.0 \times 5.0 = 150000$. Dividing the sum of *yes* and *no* volumes by the *total* volume, we obtain the percentage of the cases for which the analysis gives a definite answer. We will call this percentage the precision of the analysis.

Varying the minimum width allowed First of all, we run the analysis modifying the *minimum width allowed* (MWA) parameter and we reported the results of these tests in Table 1. We can clearly see the trade-off between performance and precision.

Table 1. Varying the minimum width allowed (MWA)

MWA	Time (sec.)	*yes*+*no* volume	Precision
3	530	131934	88%
5	77	99219	66%
12	11	40625	27%
24	1	25000	17%
45	0.2	0	0%

Finding appropriate starting values In Table 2 we reported the results of a series of successive tests obtained by changing the horizontal velocity of the ball (`vx`). In particular, we made up a series of tests simulating the behavior of a developer using our analysis to debug his code. Let us suppose that we wrongly inserted a starting interval of negative values (between -120 and 0) for the horizontal velocity. The first test (# 1) shows us that the program does not work correctly, since the *no* volume is 100%. Also, to give this answer, the analysis is very quick because a low MWA (45) suffices. After that, we try (test # 2) with very high positive velocities (between 60 and 120) and we obtain (also very quickly) a 100% of positive answer: we know for sure that with these velocities the program works correctly. Now it remains to verify what happens with velocities between 0 and 60, and we try this in test # 3, where we decrease the MWA because we need more precision (the results with greater MWA were presented by the previous Section). Some values of `vx` (i.e., ≥ 31.25) ensure that the property is verified, some other values (i.e., ≤ 12.5) ensure that the property is not verified, but the ones in between are uncertain. Tests # 4 and # 5 are just double checks. So we try with a smaller MWA (3) in test # 6 on the interval [15..30]: about a quarter of the starting values produces *yes* and another quarter produces *no*. The *no* derives from low values (smaller than 18) and we confirm this in test # 7. As for medium-high values, test # 6 shows that, with a velocity greater than 25, the answer is *almost* always *yes*. It is not always yes because, with this range of velocities, the values of other variables become important to verify the property. Test # 8, in fact, shows us that velocities within 25 and 30 produce an 82% of *yes*, but a 18% of *maybe* remains. Finally, in test # 9 we modify also other two variables (with values chosen looking at the results from test # 6 and # 8) and, with such values, the answers are 100% *yes*.

After these tests, the developer of the case study is sure that horizontal velocities below 18 will certainly not make the program work. On the other hand, values greater than 30 certainly make the program work. For values between 25 and 30, other variable values must be changed (`px` and `py`) to make the program work correctly. Making some other tests, we could also explore what happens with values between 18 and 25.

Discussion In this scenario, we ran the analysis by manually changing the initial values of program variables. Notice that this process could be automatized. This

Table 2. Varying the horizontal velocity (vx)

Test	vx interval	MWA	Time (sec)	Answer	Comment
# 1	[-120 .. 0]	45	1	$no = 100\%$	With negative values the answer is always no.
# 2	[60 .. 120]	45	0.2	$yes = 100\%$	With very high positive values the answer is always yes.
# 3	[0 .. 60]	5	77	$yes = 45\%$ $no = 21\%$	Uncertainty. High values (≥ 31.25) imply yes, low values (≤ 12.5) imply no.
# 4	[0 .. 15]	24	0.5	$no = 100\%$	Double check on low values: answer always no.
# 5	[30 .. 60]	5	30	$yes = 100\%$	Double check on medium-high values: answer always yes.
# 6	[15 .. 30]	3	526	$yes = 27\%$ $no = 25\%$	Uncertainty. Low values (≤ 18) imply no, for high values (≥ 25) depends also on other variables.
# 7	[15 .. 18]	5	7	$no = 100\%$	Double check on medium-low values: answer always no.
# 8	[25 .. 30]	3	164	$yes = 82\%$ $maybe = 18\%$	Double check on medium-high values: answer almost always yes. In this case, also values of other variables influence the result.
# 9	[25 .. 30]	5	1	$yes = 100\%$	Modified also py ([40 .. 50]) and px ([5 .. 10]). Answer always yes.

process can be highly interactive, since the tool could show to the user even partial results while it is automatically improving the precision by adopting narrower intervals on the *maybe* portion as described by Algorithm 1. In this way, the user could iterate the process until it finds suitable initial values.

The execution times obtained so far underline that the analysis is efficient enough to be the basis of practical tools. Moreover, the analysis could be parallelized by running in parallel the computation of the semantics for each initial hypercube: exploiting several cores or even running the analysis in the cloud, we could further improve the efficiency of the overall analysis.

7 Related work

Various numerical domains have been studied in the literature, and they can be classified with respect to a number of different dimensions: finite (e.g., *Sign*) versus infinite (e.g., *Intervals*) height, relational (e.g., Octagons [19]) versus non-relational (e.g., *Intervals*), convex (e.g., Polyhedra [10]) versus possibly non-convex (e.g., donut-like domains [13]). Hypercubes track disjunctive information relying on Intervals. Similarly, the powerset operator [12] allows one to track disjunctive information, but the complexity of the analysis grows up exponentially. Instead, we designed a specific disjunctive domain that reduces the practical complexity of the analysis by adopting indexes and offsets.

Noticeable efforts have been put both to reduce the loss of precision due to the upper bound operation, and to accelerate the convergence of the Kleene iterative algorithm [15, 22, 21, 4], but they do not track disjunctive information.

The trace partitioning technique designed by Mauborgne and Rival [18] provides automatic procedures to build suitable partitions of the traces yielding to a refinement that has great impact both on the accuracy and on the efficiency

of the analysis. This approach tracks disjunctive information, and it works quite well when the single partitions are carefully designed by an expert user. Unluckily, given the high number of hypercubes tracked by our analysis, this approach is definitely too slow for the scenario we are targeting.

Our spatial representation and width adjustment resembles the hierarchical data-structure of quadtrees in [17]. However, this paper contains only a preliminary discussion of the quadtree domain, and as far as we know it has not been further developed nor applied. Moreover, their domain is targeted to analyse only machine integers and the width is the same in each spatial axis.

Our self-adaptive parametrization of the width shares some common concepts with the CounterExample Guided Abstraction Refinement (CEGAR) [8]. CEGAR begins checking with a coarse (imprecise) abstraction of the system and progressively refines it, based on spurious counterexamples seen in prior model checking runs. The process continues until either an abstraction proves the correctness of the system or a valid counterexample is generated.

[14] introduced the Boxes domain, a refinement of the Interval domain with finite disjunctions: an element of Boxes is a finite union of boxes. Each value of Boxes is a propositional formula over interval constraints and it is represented by the Linear Decision Diagrams data structure (LDDs). Note that the size of an LDD is exponential in the number of variables. We use a fixed width and a fixed partitioning on each hypercube dimension, while they do not employ constraints of this kind. In addition, Boxes uses a specific abstract transformer for each possible operation (for example, distinguishing $x = x + v$, $x = a \times x$, $x = a \times y$ and also making assumptions on the sign of constants) while our definitions are more generic. Finally, Boxes' implementation is based on the specific data structure of LDDs and cannot be extended to other base domains.

If on the one hand Parametric Hypercubes have been tailored to Computer Games Software applications, on the other hand some of their features may also be applied to other contexts. In particular, our definition of Computer Games Software applications (i.e., an infinite reactive loop, a complex state space with many real-valued variables, and strong dependencies among variables) exactly matches that of real-time synchronous control-command software (found in many industries such as aerospace and automotive industries). Hybrid systems and hybrid automata have been widely applied to verify this software. The formal analysis of large scale hybrid systems is known to be a very difficult process [1]. In general, existing approaches suffer from performance issues or limitations on the property to prove, on the shape of the program, etc. For instance, [7] deals a simpler example than ours (a bouncing ball with only vertical motion) and in their benchmarks the variable space is quite limited: the velocity is a fixed constant, and the starting position varies only between 10 and 10.1. Instead, our Hypercubes can deal with velocities and positions bound inside any intervals of values. Also in [5] the variable space is more restricted than in our approach. In addition, this analysis returns an abstraction of the final state of the program, while we also give information about which starting values are responsible for the property verification and which not. [16] presents an application of the abstract

interpretation by means of convex polyhedra to hybrid systems. This work is focused on a particular class of hybrid systems (*linear* ones), and it is able to represent only convex regions of the space, since it employs the convex hull approximation of a set of values. [2] presents algorithms and tools for reachability analysis of hybrid systems by relying on predicate abstraction and polyhedra. However, this solution suffers from the exponential growth of abstract states and relies on expensive abstract domains. Finally, [20] concerns safety verification of non-linear hybrid systems, starting from a classical method that uses interval arithmetic to check whether trajectories can move over the boundaries in a rectangular grid. This approach is similar to ours in the data representation (boxes). However, they do not employ any concept of offset, their space partitioning is not fixed and the examples they experimented with cover a very limited variable space.

8 Conclusions and future work

In this paper we presented Parametric Hypercubes, a disjunctive non-relational abstract domain which can be used to analyse physics simulations. Experimental results on a representative case study show the precision of the approach. The performance of the analysis makes it feasible to apply it in practical settings.

Note that our approach offers plenty of venues in order to improve its results, thanks to its flexible and parametric nature. In particular, we could: (i) increase the precision by intersecting our hypercubes with arbitrary bounding volumes which restrict the relationships between variables in a more complex way than the offsets presented in Section 5; (ii) increase the performance of Algorithm 1 by halving the widths only on some axes, chosen through an analysis of the distribution of hypercubes in the *yes,no,maybe* sets; and (iii) study the derivative with respect to time of the iterations of the main loop in order to define temporal trends to refine the widening operator. In addition, our domain is modular w.r.t. the non-relational abstract domain adopted to represent the hypercube dimensions. By using other abstract domains it is possible to track relationships between variables which do not necessarily represent physical quantities.

References

1. R. Alur, T.A. Henzinger, G. Lafferriere, and G.J. Pappas. Discrete abstractions of hybrid systems. *Proceedings of the IEEE*, 88(7):971–984, 2000.
2. Rajeev Alur, Thao Dang, and Franjo Ivancic. Reachability analysis of hybrid systems via predicate abstraction. In *Hybrid Systems: Computation and Control, Fifth International Workshop, LNCS 2289*, pages 35–48. Springer-Verlag, 2002.
3. Gianluca Amato and Francesca Scozzari. The abstract domain of parallelotopes. *Electronic Notes Theoretical Computer Science*, 287:17–28, 2012.
4. Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. Widening operators for powerset domains. *STTT*, 9(3-4):413–414, 2007.
5. Olivier Bouissou. Proving the correctness of the implementation of a control-command algorithm. In *Static Analysis, 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings*, volume 5673 of *Lecture Notes in Computer Science*, pages 102–119. Springer, 2009.

6. Olivier Bouissou. From control-command synchronous programs to hybrid automata. In *Analysis and Design of Hybrid Systems (ADHS'12)*, June 2012.
7. Olivier Bouissou, Samuel Mimram, and Alexandre Chapoutot. Hyson: Set-based simulation of hybrid systems. In *Proceedings of the 23rd IEEE International Symposium on Rapid System Prototyping, RSP 2012, Tampere, Finland*, pages 79–85, 2012.
8. Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification (CAV), 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2000.
9. P. Cousot. The calculational design of a generic abstract interpreter. In *Calculational System Design*. NATO ASI Series F. IOS Press, Amsterdam, 1999.
10. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of POPL '78*. ACM Press, 1978.
11. D.H. Eberly. *Game Physics*. Interactive 3D technology series. Elsevier Science, 2010.
12. Gilberto Filé and Francesco Ranzato. The powerset operator on abstract interpretations. *Theor. Comput. Sci.*, 222(1-2):77–111, 1999.
13. Khalil Ghorbal, Franjo Ivancic, Gogul Balakrishnan, Naoto Maeda, and Aarti Gupta. Donut domains: Efficient non-convex domains for abstract interpretation. In *Proceedings of VMCAI '12*, pages 235–250. Springer-Verlag, 2012.
14. Arie Gurfinkel and Sagar Chaki. Boxes: A symbolic abstract domain of boxes. In *Static Analysis - 17th International Symposium, SAS 2010, Perpignan, France, September 14-16, 2010. Proceedings*, volume 6337 of *Lecture Notes in Computer Science*, pages 287–303. Springer, 2010.
15. Nicolas Halbwachs, David Merchat, and Laure Gonnord. Some ways to reduce the space dimension in polyhedra computations. *Formal Methods in System Design*, 29(1):79–95, 2006.
16. Nicolas Halbwachs, Pascal Raymond, and Yann eric Proy. Verification of linear hybrid systems by means of convex approximations. pages 223–237. Springer-Verlag, 1994.
17. Jacob M. Howe, Andy King, and Charles Lawrence-Jones. Quadtrees as an abstract domain. *Electronic Notes in Theoretical Computer Science.*, 267(1):89–100, 2010.
18. Laurent Mauborgne and Xavier Rival. Trace partitioning in abstract interpretation based static analyzers. In *Proceedings of ESOP '05*, pages 5–20. Springer-Verlag, 2005.
19. A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 2006.
20. Stefan Ratschan and Zhikun She. Safety verification of hybrid systems by constraint propagation based abstraction refinement. In *Proceedings of HSCC 2005*, volume 3414 of *Lecture Notes in Computer Science*, pages 573–589. Springer, 2005.
21. Sriram Sankaranarayanan, Franjo Ivancic, Ilya Shlyakhter, and Aarti Gupta. Static analysis in disjunctive numerical domains. In *Proceedings of SAS '06*, pages 3–17. Springer-Verlag, 2006.
22. Yassamine Seladji and Olivier Bouissou. Fixpoint computation in the polyhedra abstract domain using convex and numerical analysis tools. In *Proceedings of VMCAI '08*, pages 149–168. Springer-Verlag, 2013.