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. Application domains like Computer Games Software are an interesting workbench to stress the trade-off between accuracy and efficiency of abstract domains for static analysis, in the Abstract Interpretation framework. Game software deeply relies on physics simulations, which are particularly demanding: due to the large amount of interleaving floating point variables, the numerical domains already studied in the literature either lack accuracy too much, or their use becomes unfeasible due to the exponential cost of abstract operations.
In this paper, we discuss the domain of 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 a `while` loop which computes the numerical integration over time (i.e., the inductive step of the simulation). The `while` loop is executed in real time, and its condition is `true` for the whole duration of the program: we cannot know beforehand the number of loop iterations which will be executed by the program, because the loop will be 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. Simulations usually deal with positions, velocities, and accelerations. Finally, 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.

In order to prove statically any interesting property on such programs, we need to track precisely these relationships. Moreover, we would like to track additional functional information like the reachability of some configurations of the state. For example, we may want to ensure that a rocket reaches a stable orbit instead of falling to the ground, or that a bouncing ball arrives to the end of the screen. We discuss in depth a case study in Section 3.

Traditional approaches to static analysis do not seem to properly address these properties on this kind of programs. On the one hand, non-relational domains guarantee efficient analyses, but they are usually too approximate. On the other hand, the computational cost of sophisticated relational domains like Polyhedra [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 main novelties 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 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 shows how to improve the performance and precision of the analysis. Section 7 contains the experimental results of our analysis applied to the case study of Section 3. Section 8 presents the related work and concludes.

2 Language syntax

In this Section we present the language syntax supported by our analysis.

Let $\mathcal{V}$ be a finite set of variables. Let $\mathcal{I}$ be the set of all real-valued intervals (excluding $\mathbb{R}$ itself). We define inductively the syntax of programs in Figure 1.

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

Fig. 1. Syntax

We focus on programs dealing with mathematical computations over real-valued variables. Therefore, we consider expressions built through the most common mathematical operations (sum, subtraction, multiplication, and division). An expression can be a constant value ($c \in \mathbb{R}$), a non-deterministic value in an interval of values ($I \in \mathcal{I}$), or a variable ($V \in \mathcal{V}$). We also consider boolean conditions built through the comparison of two 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 of statements.

3 The case study of bouncing balls

As a case study, consider a program which generates a bouncing ball on the screen. The ball starts at the left side of the screen (even though the exact starting 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 the horizontal component of the velocity is always positive. Whenever the ball reaches the "ground" (i.e., 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 *Property 1*). The code of the program which simulates the creation and movement of the ball is reported in Listing 1.1.

Listing 1.1. Case study: bouncing-ball code

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

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 meanings of the variables are as follows:

- (`px`, `py`) represents the current position of the ball in the screen. The initial position of the ball is generated randomly. We only know that the ball is generated in proximity of the left side of the screen (since the x-coordinate is between 0.0 and a low value such as 10.0).
- (`vx`, `vy`) represents the current velocity of the ball. The initial velocity is generated randomly as well. Note that the horizontal velocity is always positive, because we want to throw the ball towards the right of the screen. The vertical velocity, instead, is negative, because we throw the ball downwards.
- `dt` represents the time interval between iterations of the loop. This value is constant and known at compile time. We consider a simulation running at 20 frames per second: this means that `dt` $= 1/20 = 0.05$.
- `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 world variables, that is, the ball position and velocity. In particular, if the ball is in the air (i.e., its vertical position is greater or equal to 0.0), its position is updated according to the rule of uniform linear motion. The vertical velocity is also updated to take into account the force of

gravity, while the horizontal one remains unmodified. Otherwise, the ball touches the ground[4] and it bounces. To simulate the bouncing, we update the horizontal position as usual, and we force the vertical position to zero. Moreover, the vertical velocity is inverted (instead of going downwards, the ball must go upwards) and decreased, along with the horizontal velocity, through the constant factor `k`, to consider the force which is lost in the impact with the ground.

Verifying properties on this kind of programs can have interesting practical impact also on more complex programs, since it is a basic physics simulation which can be used in many contexts [4]. For instance, consider the case where some game entity 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 pseudo code of the main method of this simulation is shown in Listing 1.2, where `updateBall(b)` is a function which updates the ball `b` according to the physics of the simulation (i.e., the body of the while loop of Listing 1.1) and `generateNewBall()` is a function which creates a new ball (with the values of the initialization of Listing 1.1).

Listing 1.2. Bouncing ball generation

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

If we verify *Property 1* on Listing 1.1, we are sure that a single ball will have exited the screen after T seconds. We also know that, in Listing 1.2, we generate one ball each `creationInterval` seconds. This means that, having verified *Property 1*, we can guarantee in Listing 1.2 that *a maximum of* $\lceil \frac{T}{\texttt{creationInterval}} \rceil$ *balls will be on the screen at the same time*. Such information is useful for performance reasons (crucial in a game), since each ball requires computations for its rendering and updating.

The theoretical interest of this case study lies in the fact that non-relational or non-disjunctive approaches 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

[4] We must wait that the position of the ball is *lower* than zero before making it bounce, otherwise a ball could bounce before having actually touched the ground. The ball could disappear from the screen for a bit, but, considering the frame-rate of modern games, this artifact is almost not perceived by the user.

possibly goes to zero, the analysis is no more able to distinguish which branch of the `if` − `then` − `else` to take. In this case, the lub operator makes the vertical velocity interval quite wider, 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

In this Section we introduce the Parametric Hypercubes abstract domain $\mathcal{H}$.

Intuitively, an abstract state of our domain 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. Each dimension of the space represents the possible values that the corresponding variable of the program can assume. The concrete trace of a program is a sequence of points in such space (one for each state of the trace). The hypercubes of our $\mathcal{H}$ domain are *volumes* in the same $|\mathsf{Vars}|$-dimensional space. Each side of the hypercube is the concretization of the abstract value of the corresponding variable, and thus it corresponds to a set of values in that dimension of the space. The concretization of an hypercube is the set of all the points contained in its volume. A state in $\mathcal{H}$ is composed by a set of hypercubes: its concretization is the union of all the volumes of its hypercubes. In this way we track disjunctive information.

4.1 Lattice structure

An abstract state of $\mathcal{H}$ is made by a *set* of hypercubes. Each hypercube is represented by a tuple of abstract values. The dimension of these tuples is equal to the number of program variables: this means that each variable is associated to a given item of the tuple (i.e., to a specific side of the hypercube). Consider for instance a program in which $\mathsf{Vars} = \{x_1, x_2\}$. In this case, the hypercubes of $\mathcal{H}$ are 2D-rectangles. In particular, the two sides of a single hypercube are two abstract values, one for x_1 and one for x_2.

A priori, our approach is modular w.r.t. the non-relational abstract domain we adopt to approximate the values of single variables inside an hypercube. We abstract floating-point variables through intervals of real values. We adopt set

of hypercubes to improve the precision of the analysis and to track disjunctive information. This is useful, for instance, when the values of a variable are clustered in different ranges: instead of having a very big interval to cover them all (and which would cover also a lot of invalid values), we use two (or more) smaller intervals. Since it would be particularly expensive to perform all the lattice operators pointwisely, we partition the possible values into intervals of fixed width. As an example, suppose that the initial vertical velocity of the balls of our case study ranges between 50.0 and 60.0 or between -60.0 and -50.0. A single interval would approximate these values with $[-60.0, 60.0]$, while with our approach we track two intervals, $[-60.0, -50.0]$ and $[50.0, 60.0]$ (with fixed width 10.0), which distinguish between balls thrown downwards and balls thrown upwards. 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. 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.

Since a single hypercube is a tuple of intervals (one for each side), 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. 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. In Section 6 we will present a variation of our analysis which computes automatically the widths, adapting them recursively and freeing the user from the need to specify values for them. For now, just 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). Each width w_i is a floating point number that 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 $[m \times w_i, (m+1) \times w_i]$.

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. The area of each rectangle is $10.0 \times 25.0 = 250.0$. We can draw such rectangles on the Cartesian plan, where the horizontal axis represents the values which `px` can assume, and the vertical axis represents those of `py`. Each side of a hypercube is identified by an integer index, representing the interval of values associated to that side. A 2D hypercube is then uniquely identified by a pair of integers. For instance, the hypercube $h_1 = (0, 1)$ says that `px` $\in [0.0, 10.0]$ and `py` $\in [25.0, 50.0]$, while the hypercube $h_2 = (0, 0)$ associates `px` to $[0.0, 10.0]$ and `py` to $[0.0, 25.0]$. In Figure 2 we can see the graphical representation of the two hypercubes associated to the initialization of the case study (i.e., h_1 and h_2). The two axes of the Cartesian plan are split in correspondence of multiples of 10.0 (x-axis) and 25.0 (y-axis). In Figure 3, instead, we depict the hypercubes obtained after executing the first iteration of the `while` loop. Note that the hypercubes are now six. The ball is moving towards the right of the screen and

is going downwards: this is coherent with the fact that the horizontal velocity is certainly positive (between 0.0 and 60.0), while the vertical velocity is certainly negative (between -30.0 and -25.0).

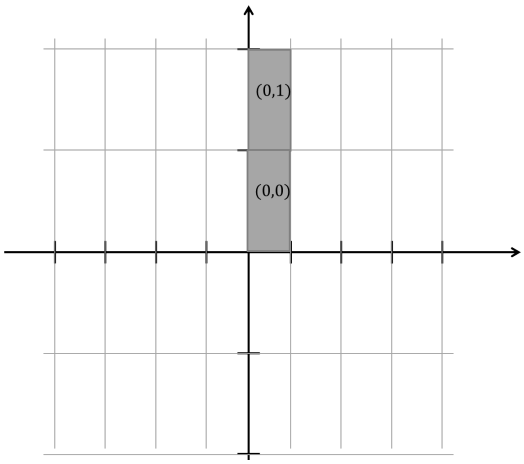

Fig. 2. The abstract state of the case study after the initialization of the variables (focusing the attention only on $\mathtt{px}, \mathtt{py}$, when their widths are, respectively, 10.0 and 25.0)

We now define formally 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: the partial order is defined through set inclusion, the lub and glb are set union and set intersection, respectively, while bottom and top are the empty set and the set containing all possible n-dimensional hypercubes, respectively. Formally, the lattice definition is $\langle \wp(\mathbb{Z}^n), \subseteq, \cup, \cap, \emptyset, \mathbb{Z}^n \rangle$.

Lemma 1. *$\langle \wp(\mathbb{Z}^n), \subseteq, \cup, \cap, \emptyset, \mathbb{Z}^n \rangle$ is a complete lattice.*

4.2 Concretization function

We denote by $\overline{\mathsf{A}}$ the non-relational abstract domain on which our analysis is parameterized, and by n the number of variables of the program, where $n = |\mathsf{Vars}|$.

$$\gamma_{\overline{\mathsf{Val}}} : \wp(\mathbb{Z}^n) \to \wp(\mathbb{R}^{\mathsf{Vars}})$$
$$\gamma_{\overline{\mathsf{Val}}}(\overline{\mathsf{V}}) = \{\sigma : \exists v \in \overline{\mathsf{V}} : \forall i \in [1..n] : \sigma_i \in \gamma_{\overline{\mathsf{A}}}(\overline{getAbsValue}_v(i))\}$$

where: (i) $\sigma \in \mathbb{R}^{\mathsf{Vars}}$ and $\sigma_i \in \mathbb{R}$ denotes the i-th element of the tuple σ, (ii) $\gamma_{\overline{\mathsf{A}}} : \overline{\mathsf{A}} \to \wp(\mathbb{R})$ is the concretization function of abstract values of the non-relational abstract domain on which our analysis is parameterized, and (iii)

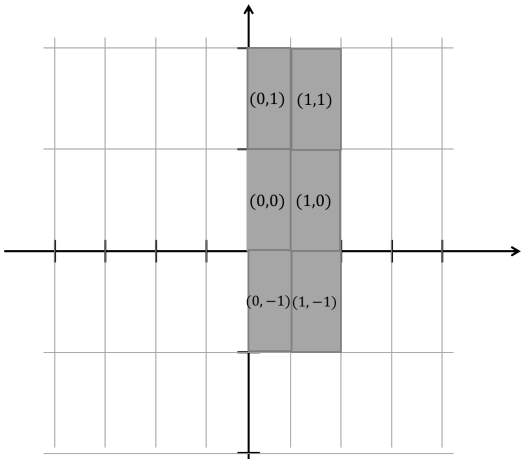

Fig. 3. The abstract state of the case study after the first iteration of the loop (focusing the attention only on `px`, `py`, when their widths are, respectively, 10.0 and 25.0)

$\overline{getAbsValue}_v : \mathbb{N} \to \overline{\mathsf{A}}$ is the function that, given an integer index, returns the abstract value (in the domain $\overline{\mathsf{A}}$) which corresponds to that index inside the tuple v.

$\gamma_{\overline{\mathsf{Val}}}$ concretizes a set of hypercubes to a set of vectors of n floating point values.

$$\gamma_{\mathcal{H}} : \wp(\mathbb{Z}^n) \to \wp(\mathsf{Vars} \to \mathbb{R})$$
$$\gamma_{\mathcal{H}}(\overline{\mathsf{V}}) = \{[\mathtt{x} \mapsto \mathsf{r}(\overline{varIndex}(\mathtt{x})) : \mathtt{x} \in \mathsf{Vars}] : \mathsf{r} \in \gamma_{\overline{\mathsf{Val}}}(\overline{\mathsf{V}})\}$$

$\gamma_{\mathcal{H}}$ simply transforms the vectors returned by $\gamma_{\overline{\mathsf{Val}}}$ into concrete environments relying on the function $\overline{varIndex} : \mathsf{Vars} \to \mathbb{N}$, that, given a variable, returns its index in the elements of $\mathcal{H}$.

4.3 Widening operator

The domain described so far does not ensure the convergence of the analysis. In fact, a `while` loop may add new hypercubes with increased indices at each iteration, and the dimension of the abstract state (i.e., the hypercubes set) would increase at each iteration without converging. Thus, we need a way to force the convergence of the analysis. Given our abstract state representation, 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, and the resulting domain has finite height. Thus, in this scenario, the widening operator coincides with the lub operator, i.e., set union.

This approach may seem too rough since we establish the bounds of intervals before running the analysis. However, this allows us to control the number of

possible intervals in our hypercubes, and this is particularly important for the efficiency of the overall analysis. In addition, when analysing physics simulations we can use the initialization of variables and the property we want to check 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.

Observe that more sophisticated widening operators could be used as an alternative to the adopted solution described above, but this could affect the performance of the resulting analysis.

5 Abstract semantics

For the most part, the abstract semantics applies existing semantic operators of boxed Intervals [2].

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

This semantics is used to define the abstract semantics $\mathbb{B}$ of Boolean comparisons. Given a hypercube and a Boolean comparison $E_1 < bop > E_2$ where $< bop > \in \{\geq, >, \leq, <, \neq\}$, this semantics 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 conditions 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.

Note that in this way we lose some precision. For instance, imagine that in a given hypercube we know that $\mathtt{x} \in [0..5]$ (because the fixed width of intervals associated to `x` is 5), and we check if $\mathtt{x} \leq 3$. The answer of $\mathbb{B}$ will be $\top$ and, if the condition was inside an `if` statement, this hypercube will be used to compute the semantics of both the branches. Indeed, we would know that $\mathtt{x} \in [0..3]$ in the `then` branch, and $\mathtt{x} \in [4..5]$ in the else branch. Nevertheless, we cannot and do not want to track this information in our hypercube, since the width of the interval associated to `x` is 5, and fixed widths are a key feature in order to obtain an efficient analysis. We will present (Section 6.2) how we can recursively modify the widths of the analysis to improve precision in these cases.

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 $\mathtt{x} = \mathtt{exp}$ is to (i) obtain the interval representing the right part ($\mathbb{I}[\![\mathtt{x} = \mathtt{exp}, \sigma]\!] = [m..M]$), and (ii) assign it in the current state. This approach does not necessarily produce a single hypercube, since we could assign an interval which width is greater than the fixed width of the assigned variable, or that covers several other intervals belonging to different partitions. In this case, 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 keeping the fixed widths of the intervals, which is a key feature of our domain in order to obtain an efficient analysis.

6 Tuning the analysis

The aim of this Section is to explain: (i) how to initialize the hypercubes set at the beginning of the analysis, (ii) how to select the interval widths in the hypercubes, and (iii) how to improve the precision of the analysis by using offsets to identify sub-volumes inside the hypercubes.

6.1 Initialization of the analysis

Before starting the analysis we have to determine the hypercubes dimensionality, i.e., the number of sides each hypercube will have. To do this, we must find all the variables of the code which are not constants. We require the program to initialize all non-constant variables at the beginning of the program. Note that physics simulations, like our case study, satisfy this requirement because they are made up by an initialization of all variables, followed by a `while` loop which contains the core of the program (i.e., the update of the simulated world). Otherwise, we can consider a dummy initialization (i.e., 0.0) for all variables which are not initialized at the beginning of the program. The actual initialization of the variables will be treated as a normal assignment, without any loss of precision. 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, $\mathcal{H} = \bigtimes_{V \in Vars} \alpha(V)$.

As an example, consider the initialization in Listing 1.1 of our case study. First of all, we must identify the variables which are not constants: dt, g, k are assigned only once (during the initialization), so we do not include them in $Vars$. The set of not-constant variables is then $Vars = \{V_1 = px, V_2 = py, V_3 = vx, V_4 = vy\}$. The cardinality of the hypercubes will be $|Vars| = 4$. Suppose that the widths associated to the variables are $w_1 = 10.0, w_2 = 25.0, w_3 = 30.0, w_4 = 5.0$. Then, the abstraction of each variable is $\alpha(V_1) = \{0\}$, $\alpha(V_2) = \{0, 1\}$,

$\alpha(V_3) = \{0, 1\}$, and $\alpha(V_4) = \{-6\}$. The Cartesian product of these abstractions brings us to the following initial set of hypercubes:

$$\mathcal{H} = \{(0, 0, 0, -6), (0, 0, 1, -6), (0, 1, 0, -6), (0, 1, 1, -6)\}$$

6.2 Width choice

The choice of the interval widths is quite important, because it influences both the precision and efficiency of the analysis. The widths influence the granularity of the space partitioning with respect to each variable. 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.

The width selection can be automatized. We implemented a recursive algorithm which adjusts the widths automatically, starting with bigger ones and then decreasing them only in the portions of space where it is really needed, to avoid compromising the performance.

We start with wide intervals (i.e., coarse precision, but fast results) and we run the analysis for the first time. We track, for each hypercube of the final abstract state, the initial hypercubes (*origins*) from which it is derived.

At the end of the analysis, we check, for each hypercube of the final set, if the desired property is verified. We associate to each origin its final result by merging the results of its derived final hypercubes: 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. We can partition the starting hypercubes set with respect to this criterion.

We run the analysis again, but *only* on the origins which did not give a definite answer. To obtain more precise results in this specific space portion, the analysis is now run with halved widths. Note that this step is only performed until we reach a specific threshold, i.e., the *minimum width* allowed for the analysis. This parameter can be specified by the user (together with the starting widths). The smaller this threshold is, the more precise (but slower) the analysis becomes.

At the end of this recursive process, we obtain three partitions of the variable space: a set of starting hypercubes which certainly verify the property (*yes* set), a set of starting hypercubes which certainly do not verify the property (*no* set), and a set of starting hypercubes which, at the minimum width allowed for the analysis, still do not give a definite answer (*maybe* set). This means that the analysis tells us which initial values of the variables bring us to verify the property 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.

The overall analysis takes as input the starting width, the minimum width allowed and the set of starting hypercubes (obtained from the initialization of

Algorithm 1 The width adjusting recursive algorithm

```
function ANALYSIS(currWidth, minWidth, startingHypercubes)
    return (yes ∪ yes', no ∪ no', maybe')
    where
    (yes, no, maybe) = hypercubesAnalysis(currWidth, startingHypercubes)
    if currWidth/2.0 ≥ minWidth then
        (yes', no', maybe') = Analysis(currWidth/2.0, minWidth, maybe)
    else
        (yes', no', maybe') = (Set.empty, Set.empty, maybe)
    end if
end function
```

the program as described in Section 6.1). It executes the analysis on such data with the function `hypercubesAnalysis`, which returns three sets of hypercubes (`yes`, `no`, `maybe`) with the meaning explained above. Then, if the halved width is still greater than the minimum one allowed, the algorithm performs a recursive step by repeating the analysis function only on the `maybe` hypercubes set (with halved width).

Note that the three final hypercubes sets (the *yes*,*no*,*maybe* partitions) will contain hypercubes of different sizes: this happens because each hypercube can come from a different iteration of the analysis, and each iteration is associated to a specific hypercube size. A certain portion of the variable space could give a definite answer even at coarse precision (for example, when the horizontal velocity of the ball is sufficiently high, the values of other variables do not matter so much), while another portion could need to be split in much smaller hypercubes to give interesting results.

6.3 Offsets

There is still space for improving accuracy in the domain introduced so far. 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 the initial interval associated to `x` was $[0.0, 1.0]$, after the first iteration we would obtain two intervals ($[0.0, 1.0]$ and $[1.0, 2.0]$) because the resulting interval would be $[0.01, 1.01]$, which spans over two fixed-width intervals. For the same reason, after the second iteration we would obtain three intervals ($[0.0, 1.0]$, $[1.0, 2.0]$ and $[2.0, 3.0]$) and so on: at each iteration we add one interval.

In order to overcome these situations, we may further improve the definition of our domain: each hypercube is augmented with additional information, which limits the admissible values inside it. In particular, 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 $[(i \times w_i) + o_m, (i \times w_i) + o_M]$. If $o_m = 0$ and $o_M = w_i$, the sub-interval corresponds to the whole fixed-width interval.

An element of our abstract domain is now stored (instead as a set of hypercubes) 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). Before, an abstract state was defined by $H = \{h : h \in \mathbb{Z}^{|Vars|}\}$. 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. Each hypercube is associated to a tuple of offsets and each tuple of offsets has one offset for each program variable.

The least upper bound between two abstract states ($M = M_1 \sqcup M_2$) is then defined as follows: $dom(M) = dom(M_1) \cup dom(M_2)$, 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.

We also have to modify the abstract semantics to accommodate this change. 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 becomes

$$\begin{aligned} assign(h, V_i, [a..b]) = \{ & \\ & h[i \mapsto (m, o_m, o_M)] : [m \times w_i..(m+1) \times w_i] \cap [a..b] \neq \emptyset \wedge \\ & o_m = \begin{cases} 0 & \text{if } a \leq (m \times w_i) \\ a - (m \times w_i) & \text{otherwise} \end{cases} \wedge \\ & o_M = \begin{cases} w_i & \text{if } b \geq ((m+1) \times w_i) \\ b - (m \times w_i) & \text{otherwise} \end{cases} \\ & \} \end{aligned}$$

where h is a hypercube, V_i is the assigned variable, and $[a..b]$ is the interval we are assigning. 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 again the previous example (`x = x + 0.01`, with 1.0 as width of `x` and [0..1] as initial value of `x`): after the first iteration we would return the two intervals $[0.0, 1.0]$ 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 (instead of three), with the offsets changed to $[0.02, 1.0]$ and $[1.0, 1.02]$. You can see that hypercubes proliferation is not an issue any more. When an offset moves out of the fixed interval inside which it should reside, we remove the corresponding hypercube from the set. In our example, after 100 iterations the first of the two intervals ($[0.0, 1.0]$) would be removed from the hypercubes because the offset would start at 1.0.

Offsets give us much more precision in verifying properties (as we will see in the next Section), without affecting too much the performances, since we just need to store more data for each hypercube, but the cost of the single operations remains almost the same. Indeed, the overall performance is even improved in practice because in the previous approach the number of hypercubes increased exponentially at each iteration of the `while` loop, making the analysis unfeasible.

7 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 in Listing 1.1 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.

The starting values of the variables are those specified in Listing 1.1. Moreover, 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 the time frame between one iteration and the other is constant ($dt = 0.05$), we know that 5 seconds of simulation correspond to the execution of $5/0.05 = 100$ iterations of the `while` loop. To verify this property, we apply trace partitioning [10] to track one abstract state per loop iteration until the 100-th iteration. Since we want to check if the ball is out of the screen after 100 iterations of the loop, 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, and for each test we will specify only the values which are different with respect to the standard workbench data.

For each test, the analysis returns three sets of starting hypercubes: one representing the initial values of the variables which satisfy the property (*yes* set), one representing the initial values which surely do not satisfy the property (*no* set), and one representing the initial values 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. Note that the *total* volume refers to the initial variable space, i.e., to the space defined by the intervals assigned to the variables at the beginning of the program. The *yes*, *maybe*, and *no* volumes refer to the same space, since they are defined in terms of *starting* hypercubes. These volumes are all finite, since we suppose that all the variables are initialized by a bound interval, and this is always the case in Computer Games Software.

7.1 Varying the minimum width allowed

First of all, we run the analysis modifying the *minimum width allowed* (MWA) parameter, to see how the precision and performance are affected. In Table 1 we report the results of these tests.

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%

From these results, we can clearly see that the performance decreases when we set small widths, and it is instead very good on bigger ones. On the other hand, by decreasing the MWA we also gain more precision. For instance, when MWA $= 45$ we do not have any certain answer, while with MWA $= 3$ the certain answers cover the 88% of the volume space, a quite precise result.

7.2 Finding appropriate starting values

Let us see now how our analysis can help a developer to understand which starting values make the program work correctly. To make the presentation clearer, we will focus on the value of a single variable. In particular, we concentrate on

the horizontal velocity of the ball (`vx`), which (intuitively) should have the major impact on the verification of the property.

In Table 2 we reported the results of a series of successive tests, supposing we were the developer of the case study wishing to find out correct starting values for the program. 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) bring to verify the property, some other values (i.e., ≤ 12.5) bring to not verify the property, but the ones in between are uncertain. To do a double check about this data, we execute also tests # 4 and # 5, where we keep, respectively, only the low (between 0 and 15) and the high (between 30 and 60) values: in both cases the analysis is fairly quick in confirming the 100% *no* and 100% *yes*. 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* derive from low values (smaller than 18) and we confirm this in test # 7, where (with a MWA of 5 and quick execution time) we obtain a 100% of *no* answers. 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): in particular, we set the horizontal position (`px`) between 5 and 10, and the vertical position (`py`) between 40 and 50. With these 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. In particular, instead of providing a program that initializes all the variables, and checking if a given property holds, a user could give only the `while` loop that performs the physics simulation, and then try wide ranges of initial values for the variables. This will obviously lead to an analysis that, even with large

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.

widths, could quickly give definite answers for a wide part of the given starting intervals. This 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. In this way, when we choose a narrow width, we could exploit several cores or even run the analysis in the cloud, thus improving the efficiency of the overall analysis.

8 Related work and Conclusions

A number of different numerical abstract domains have been studied in the literature, and they can be classified with respect to a number of different dimensions: finite versus infinite height, relational versus non-relational, convex versus possibly non-convex, and so on.

The computational cost raises when lifting from finite non–relational domains like *Sign* or *Parity*, to infinite non–relational domains like *Intervals*, to sophisticated infinite relational domains like *Octagons* [11], *Polyhedra* [3], *Pentagons* [9], and *Stripes* [5], or to donut-like non-convex domains [7]. Moreover, when considering possibly non-convex disjunctive domains, as obtained through the powerset operator [6], the complexity of the analysis is growing (as well as its accuracy) in a full orthogonal (exponential) way.

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. Some ways to reduce the space dimension in polyhedra computations relying on variable elimination and Cartesian factoring are introduced in [8]. Seladji and Bouissou [13] designed refinement tools based on convex analysis to express the convergence of convex sets using support functions, and on numerical analysis to accelerate this convergence applying sequence transformations. On the other hand, Sankaranarayanan et al. [12] faced the issue of reducing the computational cost of the analysis using a powerset domain, by adopting restrictions based on "on the fly elaborations" of the program's control flow graph. Efficiency issues about convergence acceleration by widening in the case of a powerset domain have been studied by Bagnara et al. in [1]. All these domains do not track disjunctive information.

Finally, the trace partitioning technique designed by Mauborgne and Rival [10] 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.

The Parametric Hypercubes proposal presented in this paper can be seen as a selection and a combination of most of these techniques, tailored to get a solution that properly suits the features of Computer Games Software applications. In this scenario, we had to track precisely a lot of disjunctive information. Therefore, we needed to introduce a targeted domain $\mathcal{H}$ for physics simulations. On the other hand, some of the domain's features (like width parameter tuning, and interval offsets) may also be applied in other domains.

8.1 Future work

We observe 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 (the offsets presented in Section 6 are the simplest version of this extension); (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. Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. Widening operators for powerset domains. *STTT*, 9(3-4):413–414, 2007.
2. P. Cousot. The calculational design of a generic abstract interpreter. In *Calculational System Design*. NATO ASI Series F. IOS Press, Amsterdam, 1999.
3. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of POPL '78*. ACM Press, 1978.
4. D.H. Eberly. *Game Physics*. Interactive 3D technology series. Elsevier Science, 2010.
5. Pietro Ferrara, Francesco Logozzo, and Manuel Fähndrich. Safer unsafe code for .net. In *OOPSLA*, pages 329–346, 2008.
6. Gilberto Filé and Francesco Ranzato. The powerset operator on abstract interpretations. *Theor. Comput. Sci.*, 222(1-2):77–111, 1999.
7. Khalil Ghorbal, Franjo Ivancic, Gogul Balakrishnan, Naoto Maeda, and Aarti Gupta. Donut domains: Efficient non-convex domains for abstract interpretation. In *VMCAI*, pages 235–250, 2012.
8. 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.
9. Francesco Logozzo and Manuel Fähndrich. Pentagons: a weakly relational abstract domain for the efficient validation of array accesses. In *SAC*, pages 184–188, 2008.
10. Laurent Mauborgne and Xavier Rival. Trace partitioning in abstract interpretation based static analyzers. In *ESOP*, pages 5–20, 2005.
11. A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 2006.
12. Sriram Sankaranarayanan, Franjo Ivancic, Ilya Shlyakhter, and Aarti Gupta. Static analysis in disjunctive numerical domains. In *SAS*, pages 3–17, 2006.
13. Yassamine Seladji and Olivier Bouissou. Fixpoint computation in the polyhedra abstract domain using convex and numerical analysis tools. In *VMCAI*, pages 149–168, 2013.