Cutting the Cake Into Crumbs: Verifying Envy-Free Cake-Cutting Protocols using Bounded Integer Arithmetic

Martin Mariusz Lester[1][0000−0002−2323−1771]

University of Reading, Reading, United Kingdom m.lester@reading.ac.uk

Abstract. Fair division protocols specify how to split a continuous resource (conventionally represented by a cake) between multiple agents with different preferences. Envy-free protocols ensure no agent prefers any other agent's allocation to his own. These protocols are complex and manual proofs of their correctness may contain errors. Recently, Bertram and others [5] developed the DSL *Slice* for describing these protocols and showed how verification of envy-freeness can be reduced to SMT instances in the theory of quantified non-linear real arithmetic. This theory is decidable, but the decision procedure is slow, both in theory and in practice.
We prove that, under reasonable assumptions about the primitive operations used in the protocol, counterexamples to envy-freeness can always be found with bounded integer arithmetic. Building on this result, we construct an embedded DSL for describing cake-cutting protocols in declarative-style C. Using the bounded model-checker CBMC, we reduce verifying envy-freeness of a protocol to checking unsatisfiability of pure SAT instances. This leads to a substantial reduction in verification time when the protocol is unfair.

Keywords: fair division · constraint programming · declarative C

1 Introduction

Alice and Bob wish to divide a cake fairly. Alice cuts the cake into two slices, which she believes to be of equal size. Bob chooses the slice he believes to be bigger. Alice takes the remaining slice. Neither Alice nor Bob *envies* the other. This is true regardless of the actual relative sizes of the slices. Maybe the cake is slightly taller at one end and Alice's cut failed to take this into account, but she did not realise. Or maybe the slices have equal weight, but one slice has a strawberry on top and Bob really likes strawberries, so he chose that slice. In any case, Alice believes that both slices are equal, so is happy with whichever slice was left; Bob believes that he chose a slice better than or as good as Alice's.

This story illustrates how competing agents may amicably agree to divide a continuous resource. Problems of this kind are widely studied in economics. We represent the resource as a cake, but it could equally be land, advertising space or many other things. Similarly, the agents could (for example) be friends,

```c
#define SLICES 2
#define AGENTS 2

#include "crumbs.h"

int main() {
    slice whole, left, right;
    // Create the cake.
    whole = cake();
    // Agent 1 halves the cake.
    halve(1, left, right, whole);
    // Agent 2 picks the bigger slice.
    if (smaller(2, left, right)) {
        // Agent 2 picks the right slice.
        alloc(1, left);
        alloc(2, right);
    }
    else {
        // Agent 2 picks the left slice.
        alloc(1, right);
        alloc(2, left);
    }
    // Check that the focused agent doesn't feel envious.
    check();
}
```

Fig. 1. The well-known cut-and-choose protocol expressed in our cake-cutting protocol DSL *Crumbs*, which is embedded in C. Envy-freeness is verified using the bounded model-checker CBMC.

companies or governments. But in general, we refer to such protocols for resource allocation, where the divisions are proposed and evaluated by the competing agents, as *cake-cutting protocols*. A protocol is *envy-free* if, on termination, no agent would prefer any other agent's allocation.

The *cut and choose* protocol we described is alluded to in Genesis [10], the first book of the Bible, but only works for two agents. Selfridge and Conway independently developed a protocol for three agents in the 1960s [16]. In this protocol, slices cut by one agent are further cut into smaller slices by another agent, with each agent ultimately receiving multiple discontinuous slices. The question of whether bounded protocols for larger numbers of agents existed remained unanswered for many years. It was resolved in 2016 by Aziz and Mackenzie [3,4], who developed a protocol for any number of agents. The bound on the number of queries (markings of the cake by an agent or evaluations of the value of a slice) required by the protocol depends only on the number of agents, but is a tower of exponentials.

But with protocols being so complex, how can we specify them unambiguously and ensure they are correct? Recently, Bertram and others [5] addressed this problem through a Domain-Specific Language (DSL) called *Slice* for specifying cake-cutting protocols. Furthermore, they showed how to verify envy-freeness of a protocol using a translation from a *Slice* program into an SMT instance, which can be solved using Z3 [15].

While this approach performs acceptably in verifying correctness of the protocols they examine, it fails to find counterexamples even on trivially unfair protocols. The root cause seems to be that they encode axioms about properties of agents' valuations. This in turn requires that they target the theory of quantified real arithmetic, which is decidable, but has doubly exponential time complexity, which often manifests in practice.

We propose an alternative approach in our DSL *Crumbs* in Section 3. We observe that, if we restrict our DSL to higher-level primitives, such as dividing a cake into equally sized pieces, or comparing two pieces to see which is bigger, we need only rely on properties of valuations that follow from the standard axioms of arithmetic. Furthermore, in Section 4.2, we show that to find counterexamples to envy-freeness for a bounded number of slices, we need only consider valuations consisting of bounded integers.

Our DSL is embedded in C; see Figure 1 for an example protocol. Following the approach demonstrated by Manthey [14] and by Lester [12], we write *declarative C* specifications for our cake-cutting operations in Section 4.3, then use the bounded model-checker *CBMC* [8,11] to compile the program into a SAT instance that encodes the constraint satisfaction problem of finding an envy-inducing allocation; CBMC solves the SAT instance using a SAT solver (in our experiments, we use *Kissat* [6]). If it is unsatisfiable, the protocol is envy-free. If it is satisfiable, CBMC extracts an envy-inducing run of the protocol.

We evaluate *Crumbs* empirically and compare it with *Slice* in Section 5. We find that it is comparable in speed on correct protocols and significantly faster on incorrect protocols, where *Slice* does not terminate in a reasonable amount of time.

Our contributions are:

- the design (Section 3) and implementation (Section 4.3) of the embedded DSL *Crumbs* in declarative C;
- the novel reduction of verification of envy-freeness to bounded integer arithmetic in Section 4.1 and Section 4.2;
- the experimental evaluation in Section 5, which shows that *Crumbs* is competitive on correct protocols and faster on erroneous protocols.

The artifact supporting this paper [13] includes the implementation of *Crumbs* and scripts necessary to reproduce our experiments.

2 Preliminaries

Let us first formalise what we mean by a cake-cutting protocol.

A whole cake is represented by the real interval $[0, 1]$. A cake-cutting protocol is an algorithm that involves a fixed set of agents, A. Each agent a has a valuation function $V_a : \mathcal{P}([0,1]) \to \mathbb{R}$, which specifies how much the agent desires different allocations of cake. The input to the protocol is the set of agents' valuation functions and the output is an allocation of a portion of the cake to each agent, $P : A \to \mathcal{P}([0,1])$. A slice $s \subseteq [0,1]$ is a connected subinterval of the cake. We require that allocations consist of a finite number of slices.

A feasible allocation is one where all $P(a)$ are disjoint: $\forall a_1, a_2. P(a_1) \cap P(a_2) = \emptyset$. A complete allocation is one where the whole cake is allocated: $\bigcup P(A) = [0,1]$. An envy-free allocation is one where no agent would prefer any other agent's allocation: $\forall a_1, a_2. V_{a_1}(P(a_1)) \geq V_{a_1}(P(a_2))$.

In practice, we need to introduce some restrictions on the permitted valuation functions and allocations to develop interesting, realisable protocols. In their DSL *Slice*, Bertram and others adopt a reasonable set of assumptions on valuations, which we follow. Valuations must be normalised, non-negative, additive and continuous. In effect, this means that each agent's valuation function is fully described by a continuous, non-negative, everywhere-integrable density function v_a, such that $V_a([l,r]) = \int_l^r v_a(x)\,dx$ with $V_a([0,1]) = 1$ and $V_a([l_1,r_1] \cup [l_2,r_2]) = V_a([l_1,r_1]) + V_a([l_2,r_2])$, assuming $[l_1,r_1]$ and $[l_2,r_2]$ are disjoint.

Returning to our initial example of Alice and Bob cutting a cake, these restrictions allow Alice to express that the value of a slice is proportional to its weight, and allow Bob to express that the value of a slice increases linearly with the weight of strawberries in the slice. However, they do not allow Alice to express that she is not very hungry and has no interest in having more than $\frac{1}{3}$ of the cake, or allow Bob to express that he would rather have a whole strawberry than two half strawberries.

We also do not allow the algorithm within the protocol to access the valuation functions directly. (They might in any case not be finitely representable.) Instead, the protocol is restricted to making a finite number of queries of the agents. Agents are assumed to respond truthfully and accurately at all times.

A common query model, adopted by *Slice*, is the Robertson-Webb model. In this model, agents may be asked to *evaluate* or *mark* a slice:

evaluate Given $[l,r]$, the agent returns the value of $V_a([l,r])$.
mark Given l and v, the agent picks r such that $V_a([l,r]) = v$.

Here we diverge from *Slice*, instead choosing the higher-level primitives *split*, *trim* and *compare*:

split The agent splits a slice into a number of smaller slices that he believes have equal value.
trim Given a reference slice and a bigger slice, the agent trims the bigger slice down to the size of the reference slice.
compare The agent says which of two slices he prefers.

This selection allows us to express well-known protocols without resorting to real arithmetic, which will be important for our approach to developing an efficient verification procedure.

3 The Crumbs DSL

We now give an overview of the *Crumbs* DSL, which is embedded in C. Figure 2 shows the first part of the Selfridge-Conway procedure, which illustrates most of the language's features.

Crumbs uses two datatypes to describe cake-cutting protocols: *agents* are represented by integer IDs, starting from 1; *slices* are contiguous intervals of cake.

Crumbs provides the following operations for cutting and allocating slices of cake:

cake() Returns a single slice, consisting of a whole cake.

halve(a, l, r, w) Agent a splits the slice w into 2 equal slices, which are assigned to l and r.

third(a, l, m, r, w) Agent a splits the slice w into 3 equal slices, which are assigned to l, m and r. (Variants for splits into equal numbers of smaller pieces can be supported analogously.)

trim(a, l, r, w, s) Agent a trims the slice w into 2 slices, assigned to l and r, such that he believes l is equal to slice s. (It is an error to trim w if a believes it is smaller than s.)

alloc(a, s) Allocates slice s to agent a.

check() Checks that the allocation is envy-free.

Note that the operations that cut the cake require empty slice variables to be passed in, which are used to store the resulting slices.

Slices can be compared with the following operations:

smaller(a, l, r) Returns true if agent a believes slice l is strictly smaller (lower value) than slice r.

lteq(a, l, r) Returns true if agent a believes slice l is less than or equal to slice r.

equal(a, l, r) Returns true if agent a believes slice l is less than or equal to slice r.

remember(s) Returns a copy of s, which can be compared, but should not be allocated.

same(s1, s2) Returns true if $s1$ and $s2$ are the same slices of cake. (That is, they represent the same interval of the cake, not two distinct intervals with the same perceived value.)

These operations are side-effect free functions.

For convenience, the following operations for sorting and swapping slices are also defined:

sort2(a, s1, s2) Agent a rearranges slices $s1$ and $s2$, so that he believes they are sorted in increasing order.

sort3(a, s1, s2, s3) Agent a rearranges slices $s1$, $s2$ and $s3$, so that he believes they are sorted in increasing order. (Variants for sorting larger numbers of slices can be supported analogously.)

```
#define SLICES 4
#define AGENTS 3

#include "crumbs.h"

int main() {
    slice whole, a, b, c;
    whole = cake();
    third(1, a, b, c, whole); // P1 divides cake into 3.

    sort3(2, c, b, a); // P2 thinks largest piece is A.

    if(equal(2, a, b)) {
        sort3(3, c, b, a);
        alloc(3, a);    // If P2 thinks 2 biggest parts equal,
        sort2(2, c, b); // P3, P2 and P1 choose a piece in order.
        alloc(2, b);
        alloc(1, c);
    }
    else {
        slice a1, a2;          // P2 trims A to match B.
        trim(2, a1, a2, a, b); // Call trimmed piece A1.

        slice trimmed = remember(a1); // Remember A1 for later.

        sort3(3, c, b, a1); // P3 chooses a piece.
        alloc(3, a1);

        // P2 chooses a piece, then P1.
        // If P3 didn't choose trimmed piece, P2 must.
        if (same(trimmed, a1)) {
            sort2(2, c, b);
            alloc(2, b);
            alloc(1, c);
        }
        else if (same(trimmed, b)) {
            alloc(2, b);
            alloc(1, c);
        }
        else { // Trimmed piece is c.
            alloc(2, c);
            alloc(1, b);
        }
    }
    check();
}
```

Fig. 2. The first part of the Selfridge-Conway protocol in *Crumbs*. The allocation generated is envy-free, but not complete.

swap(s1, s2) Swaps slices $s1$ and $s2$.
rol(s1, s2, s3) Rotates the slices $s1, s2, s3$ left, so that $s1 \leftarrow s2$, $s2 \leftarrow s3$ and $s3 \leftarrow s1$.
ror(s1, s2, s3) Rotates the slices $s1, s2, s3$ right; inverse of rol().

These operations move slice objects between variables. They are specified solely in terms of the preceding slice manipulation and comparison operations.

As our higher-level primitives are all expressible in the Robertson-Webb model, any *Crumbs* protocol will be expressible in *Slice*. Conversely, there may be *Slice* protocols that, because of the arithmetic they perform on slice sizes and valuations, are not expressible in *Crumbs*. However, we have not encountered any reasonable protocols like this.

Crumbs is embedded in C, so the usual C control flow features, such as loops and functions, are available. Here, *Crumbs* is more expressive than *Slice* from the user's perspective, as the latter does not support loops or recursive functions. However, this does not really affect the range of protocols that can be expressed, as the underlying verification procedure requires that control flow be bounded. For example, loops must have a statically computable bound on the number of iterations possible.

4 Verifying Protocols in Crumbs

4.1 Practical Verification of Envy-Freeness

A path of execution through a cake-cutting protocol consists of a sequence of operations: agents can cut slices of cake; agents can compare slices of cake; and slices of cake can be allocated to agents.

When execution terminates, the cake has been cut into a number of slices. Each slice has a size and, for each agent, a perceived value; these must all be non-negative. Operations on the path induce further constraints on the sizes and values of the slices. When an agent cuts a slice into two smaller slices, this induces a constraint that the sizes and values of the resulting slices are non-negative and sum to the size and values of the parent slice. Furthermore, if the agent cuts a slice into child sizes that he believes to be of equal value, this induces the constraint that the agent's values of the child slices are equal. If an agent compares two slices and declares that one is smaller than another, this induces the constraint that the agent's value of one slice is smaller than the other.

A protocol is envy-free if all paths that terminate have the property that, for all agents, the sum of an agent's perceived total value of allocated slices is not less than the agent's perceived total value of any other agent's allocated slices. That is, a protocol is not envy-free only if there is a path where an agent perceives his sum of slices to be smaller than another agent's. We can verify that a protocol is envy-free by verifying that no such path exists.

To get to a practical verification procedure from here, we need two further observations. Firstly, we can verify envy-freeness for each agent separately, *focusing* on one agent at a time. When we focus on one agent, it is sound to ignore the actual sizes of any slices and the valuations of other agents. We may lose completeness if there is an envy-inducing but infeasible path, whose infeasibility depends on these ignored details, for example, if a non-focused agent compares the same pair of slices twice. However, such situations are unlikely to occur in practice, other than as a result of an implementation error. This observation means that we can safely model a slice as a pair of numbers representing an interval over the focused agent's value, rather than a pair for each agent, plus a pair to represent the actual size.

Secondly, if agents are restricted to cutting and comparing slices, but cannot declare their perceived value numerically, all constraints on a path are equalities or inequalities of sums of the focused agent's value for a subset of slices; there are no constants, other than 0 and 1 (which represent the left and right ends of the whole cake). This observation will ultimately allow us to model slice values using bounded integers, rather than as real or rational numbers.

4.2 Bounded Integer Arithmetic

So that we can avoid having to invoke an SMT solver for real arithmetic, we would like to encode our constraint problem for finding envious runs using bounded integer arithmetic. Then it can be translated into a pure SAT instance, which may be faster to solve.

For a fixed number of slices n, even with real values, there are only finitely many different possible conjunctions of inequalities between sums of values of subsets of slices. Furthermore, we can clearly model all such relationships using integers, as we can approximate reals to arbitrary precision with rationals and, as we consider only sums with a bounded number of terms, we will not encounter any behaviour that is specific to irrational numbers, such as convergence of infinite sums. Multiplying by the denominator(s) of the rational numbers will then give integers. So a bound on the size of integers we need to consider to model the same relationships as with reals clearly exists, but in order to use this fact to construct a SAT instance for verification, we need to compute it explicitly. The following theorem gives an explicit bound.

Theorem 1. *Consider a finite set $X \subseteq \mathbb{R}$ of n non-negative real numbers. There is mapping $f : X \to \mathbb{N}$ of elements of X to non-negative integers that, extended point-wise to sets, preserves the sign (positive, negative or zero) of linear combinations of elements of X with coefficients -1, 0 or 1. Furthermore, $\sum f(X) \leq \frac{1}{6}(4^n + 2)$.*

Proof. We begin by proving the existence of f, leaving the bound on $\sum f(X)$ until later. Sort the elements of X in non-decreasing order, so that $X = \{x_1, \ldots, x_n\}$ with $x_1 \leq \ldots \leq x_n$. Argue by induction over n.

Base case: $n = 0$, so $X = \emptyset$ with f being the empty mapping and $\sum f(X) = 0$.

Inductive step: Suppose we have built a suitable mapping f_k for $X_k = \{x_1, \ldots, x_k\}$. We aim to build a mapping f_{k+1} for $X_{k+1} = \{x_1, \ldots, x_k, x_{k+1}\}$.

We need an f_{k+1} that preserves sign of linear combinations of interest. We attempt to construct f_{k+1} by extending f_k. This will preserve the sign of any linear combination over X_k.

For the new linear combinations over X_{k+1}, we need to choose a value for $f_{k+1}(x_{k+1})$ somewhere between $f_k(x_k)$ (in the case that $x_k = x_{k+1}$) and $\sum f(X_k)+1$ (in the case that $\sum X_k - f(x_{k+1})$ is positive) that gives them the correct sign. If there is such an integer value available, we pick it and we are done. If not, the value we would like to assign lies between two integers, say i and $i+1$. In this case, we pick $f_{k+1}(x_{k+1}) = i + \frac{1}{2}$, then double all values assigned by f_{k+1} to restore integrality.

We now turn our attention to the bound $\sum f(X) \leq \frac{1}{6}(4^n+2)$. The construction above gives the recurrence relation $\sum f(X_0) = 0$ and $\sum f(X_{k+1}) \leq max(\sum f(X_k)+1, 4\sum f(X_k)-1)$, which solves to the required bound.

Note that, for any $Y, Z \subseteq X$, if $\sum Y \leq \sum Z$, the theorem gives us $\sum f(Y) \leq \sum f(Z)$. Thus, as a corollary, if there is an envious run of a protocol that creates at most n slices with real values, then there is an envious run with n slices with non-negative integer values with total sum at most $\frac{1}{6}(4^n+2)$. Conversely, if there is an envious run with integer values, it is trivially an envious run with real values; if we wish, we can divide by the sum of all slices to normalise the values to the range $[0,1]$.

When we use integers to measure the value of a slice, according to a focused agent, we call them *focused agent crumbs* or simply *crumbs.*

4.3 Embedding Crumbs in Declarative C

Crumbs is implemented as a C header file, to be included at the top of a C program that describes a cake-cutting protocol using the supplied datatypes and operations. Although protocols in *Crumbs* are written in an imperative style as C programs, they are not intended to be compiled and executed. Rather, through the use of verifier-level nondeterminism, assumptions and assertions, the program specifies a constraint satisfaction problem, which can be solved using the program model-checker CBMC. We refer to this style of C program, which repurposes C as a constraint programming language, as *declarative C.*

Figure 3 shows the definition of some of the primitive operations and datatypes used in *Crumbs.* Internally, the *slice* datatype is a C `struct` storing two points `l` and `r`, representing the left-hand and right-hand ends of a slice of cake, as distances from the left-hand end of the cake, measured in *focused agent crumbs.* The datatype `P` is an unsigned integer used to represent a distance, weight or value in crumbs.

The constant `MAX_RIGHT` is the bound on the number of crumbs needed for sound verification. The function `cake()` creates a new slice representing the whole cake. As the value of the right end of the slice is uninitialised, CBMC will treat it as a nondeterministic integer that can take any value within the range of its datatype. The assumption (introduced by `__CPROVER_assume`) tells CBMC that it need only consider values up to the bound.

```
// Maximum crumbs needed for this number of slices.
#define MAX_RIGHT (((1 << ((SLICES) * 2)) + 2) / 6)

// Type of a slice of cake.
struct slice {
    P l; // Left end of slice.
    P r; // Right end of slice.
};

typedef struct slice slice;

// Create a whole cake.
inline slice cake() {
    slice s;
    s.l = 0;
    __CPROVER_assume(s.r <= MAX_RIGHT);
    return s;
}

// In: left/right are new slice objects; whole is a slice
// Out: left and right are a division of whole
#define cut(left, right, whole)
    // Pick a mid-point between the ends of the slice.
    P mid;
    __CPROVER_assume(whole.l <= mid && mid <= whole.r);
    left.l = whole.l;
    left.r = mid;
    right.l = mid;
    right.r = whole.r;

// Return value of a slice according to the focused agent.
P inline value(slice s) {
    return s.r - s.l;
}

// Totals for each focused agent.
P totals[AGENTS] = {0};

// Return the total of agent X (lvalue).
#define total(X) totals[(X)-1]
```

Fig. 3. Definitions of low-level primitives in *Crumbs*. Some details have been elided for readability.

```
// According to agent a, is slice l smaller than r?
#define smaller(a, l, r) \
    ((a) == AGENT ? value(l) < value(r) : nondet_char())

// Cut a slice of cake in half.
// w is whole slice; l and r are new slice objects; a is agent cutting.
// Afterwards, l and r are slices.
#define halve(a, l, r, w) \
    cut(l, r, w);
    // If focused agent is cutting, he must think slices are same size.
    __CPROVER_assume(((a) != AGENT) || (value(l) == value(r)));

// Trim slice s1 so it is same size as slice s2, according to agent a.
// Slice s1a is the same size as s2; s1b is the trimmings.
// Requires that agent a believes s1 is at least as big as s2.
#define trim(a, s1a, s1b, s1, s2)
    assert(((a) != AGENT) || (value(s1) >= value(s2)));
    cut(s1a, s1b, s1);
    __CPROVER_assume(((a) != AGENT) || (value(s1a) == value(s2)));

// Allocate slice s to agent a.
#define alloc(a, s)
    totals[a-1] += value(s);

// Check that the focused agent doesn't feel envious of anyone else.
void inline check() {
    for (int n = 1; n <= AGENTS; n++) {
        assert(total(n) <= total(AGENT));
    }
}
```

Fig. 4. Definitions of higher-level operations in *Crumbs*. Some details have been elided for readability.

The operations described in the Section 3 are implemented using the following lower-level operations:

cut(l, r, w) Cuts the slice w into 2 slices, which are assigned to l and r.
value(s) Returns the value of slice s for the currently focused agent.
total(a) The total value of all slices allocated to agent a, according to the currently focused agent.

`cut()` merits some further explanation. Here we use verifier nondeterminism to allow a slice to be cut anywhere. The assumption ensures that the cut must be between the left and right ends of the slice.

Figure 4 shows the implementation of a representative sample of higher-level operations. As explained earlier, we need only consider the valuation of a single agent at any particular time. We call this agent the *focused agent*, represented by the constant `AGENT`. A common pattern is that we implement operations performed by other agents using nondeterminism. We can see this most simply in `smaller()`: the focused agent does not know or care whether other agents consider one slice to be smaller than another; his envy or lack thereof depends only on his own valuations.

To have an agent `halve()` a slice, we cut it nondeterministically, then, if the agent doing the cutting is focused, use an assumption to restrict the verifier to considering only paths of execution in which the agent believes the two slices are equal. The implementation of `trim()` is similar, but also uses an `assert()` to check that the agent does believe that the slice to be trimmed is bigger than the reference slice.

When slices are allocated to an agent, `alloc()` adds their value to a running total. At the end of a protocol, `check()` asserts that the focused agent does not believe any other agent has a bigger portion. If there is a path of execution that violates this assertion, CBMC will flag it as a verification failure.

4.4 Verifying Other Safety Properties

There are some other safety properties that our implementation does not verify. We do not verify whether all slices are allocated, or whether all slices are allocated at most once. (The former intentionally does not hold in some protocols.) Nor do we verify that the number of slices generated is at most `SLICES`. However, these properties can also be verified relatively cheaply using CBMC, at least at the level of accuracy comparable to a standard static analysis, by adjusting the header file to provide different definitions of the cake-cutting operations, which ignore the size of slices and instead track the number of slices.

Firstly, we need to check that slices cannot be copied, other than using `remember()`. We can do this by turning `slice` into a C++ object with private copy constructor and copy assignment operator, then type-checking the protocol as a C++ program.

Next, to check that `SLICES` is big enough, we need to add a global variable to track the number of slices created, which we increment on calls to `cut()` and compare with `SLICES` in `check()`.

To check that no slice is allocated twice, we can augment `slice()` with a flag indicating that a slice is "live", set on creation by `cake()`, preserved on child slices and cleared on parent slices slices by `cut()`, and cleared by calls to `alloc()` and `remember()`. Then we can add an assertion to forbid `alloc()` on a slice without the flag.

Finally, to check that all slices are allocated (if this is intended by the protocol), we can add a global variable to track the number of slices allocated, which we increment on calls to `alloc()`, then modify `check()` to compare this with the number actually created.

We do not explicitly verify that protocols necessarily terminate, and our definition of envy-freeness permits non-terminating protocols. However, CBMC's translation to SAT will not terminate if it cannot statically bound the number of iterations of a loop or the depth of recursive function calls, so implicitly it does verify that protocols terminate.

5 Evaluation

We evaluated our implementation of *Crumbs* experimentally by benchmarking verification of envy-freeness for the same protocols used by Bertram and others in their presentation of *Slice*. As *Crumbs* does not support some of the low-level primitives used in *Slice*, the *Crumbs* programs are necessarily different from their *Slice* counterparts, but the high-level structure and sequence of operations is similar.

All benchmarks were run on a machine running Debian GNU/Linux 12 with an Intel i5-7500 CPU at 3.40 GHz and 64 GB RAM. As our backend verifier for *Crumbs*, we used *CBMC 5.83.0* with external SAT solver *Kissat sc2022-bulky*. Code and scripts to reproduce our results are in the artifact [13] accompanying this paper.

The results of our benchmarks are shown in Table 1. For most of the protocols, verification takes around a second for both *Crumbs* and *Slice*. The full Selfridge-Conway protocol is somewhat more complicated than the others, and here we see a meaningful difference. *Crumbs* is slower than *Slice*, but still roughly comparable.

Table 1. Time taken to verify envy-freeness for cake-cutting protocols for 2–3 agents, in both *Crumbs* and *Slice*. *Crumbs* verification times combine CBMC compilation and Kissat solving; CBMC time was negligible.

	Crumbs verification time (s)				**Slice verification time (s)**		
Protocol	*Agent 1*	*Agent 2*	*Agent 3*	*Total*	*Compile*	*Z3*	*Total*
Cut-Choose	0.02	0.02	-	0.04	0.11	0.04	0.15
Surplus	0.03	0.06	-	0.09	1.14	0.04	1.18
Selfridge-Conway-Surplus	0.34	1.52	1.74	3.60	1.22	0.93	2.15
Selfridge-Conway-Full	9.84	20.38	22.83	53.05	1.20	21.71	22.91

To demonstrate our claim that our approach is good for finding counterexamples to incorrect protocols, we deliberately introduced errors to each protocol. Table 2 shows the time taken to find counterexamples to envy-freeness for the dissatisfied agent with *Crumbs*. With *Slice*, Z3 did not terminate in less than 30 minutes on any of the examples we tried. Thus it is clear that our approach performs better for disproving envy-freeness.

Verifying envy-freeness corresponds to showing that a SAT instance is unsatisfiable, which is a co-NP problem. Conversely, finding a counterexample corresponds to showing that it is satisfiable, which is an NP problem. SAT solvers are typically faster on satisfiable instances than on unsatisfiable instances of comparable problems, which explains why our approach is fast for disproving envy-freeness.

6 Related Work

The literature on fair division is broad. We surveyed some of the main results in cake cutting in Section 1. For an overview, see Procaccia's article [16]. For a sample of the breadth of the field, see the two Dagstuhl seminars on the topic [7,2].

Slice [5] verifies envy-freeness using an encoding in quantified non-linear real arithmetic, which it solves using the SMT solver Z3 [15]. Decidability of this theory was proved by Tarski using quantifier elimination. Most modern implementations use a variant of the algorithm Cylindrical Algebraic Decomposition (CAD), but its time complexity is doubly exponential. The SMT solver SMT-RAT [9] is specialised for solving problems involving real arithmetic and manages to avoid the worst-case time complexity in many cases.

Although we reduced verification of envy-freeness to integer arithmetic, the model of cake-cutting we adopted would also permit a reduction to Mixed Integer Linear Programming (MILP), which may be faster in practice. Linear inequalities over the reals can be solved using Fourier-Motzkin elimination. Although we derived our bound on the size of integers required directly, it may also be possible to find a bound through an analysis of Fourier-Motzkin elimination.

CBMC [8,11] is a bounded model-checker for C programs. It uses a variety of program transformations, such as loop unrolling and function inlining, to bound

Table 2. Time taken to find counterexamples to envy-freeness for erroneous cake-cutting protocols for 2–3 agents in *Crumbs*.

Protocol	**Agent**	**Time (s)**	**Description of error introduced**
Cut-Allocate	2	0.02	Agent 1 cuts and chooses.
Cut-Choose	2	0.02	Slices allocated wrong way round in one branch.
Surplus	2	0.05	Unsafe trim of slice smaller than reference.
S-C-S (1)	2	0.13	Allocates trimmings of slice instead of trimmed slice.
S-C-S (2)	1	0.05	Agent 2 not forced to take trimmed slice if available.
S-C-F	1	0.07	Trimmings cut by agent who took trimmed slice.

behaviour in programs and reduce program verification to SAT. Manthey [14] illustrated how to use declarative C and CBMC to generate SAT instances corresponding to a puzzle game for the SAT Competition. Meanwhile, Lester used this approach to build the constraint programming system CoPTIC [12] and the XCSP3 constraint solver Exchequer [1]. Kissat is a leading SAT solver [6]; variants of Kissat dominated the SAT Competition 2022.

7 Conclusion

We have developed and presented the embedded DSL *Crumbs* for describing cake-cutting protocols and verifying envy-freeness. Our verification procedure uses a novel encoding of envy-freeness as a constraint satisfaction problem in declarative C that requires only integer arithmetic. This enables us to implement it efficiently using the bounded model-checker for C programs CBMC, which in turn translates the verification problem into a pure SAT instance. We have evaluated *Crumbs* experimentally on a number of well-known protocols and some erroneous versions of those protocols. Verification of correct protocols was comparable in speed to the existing cake-cutting DSL *Slice*. For erroneous protocols, the verification procedure employed in *Slice* was too slow to be practical, whereas our approach was very fast.

There are further safety properties that we could verify by extending our implementation, but we leave that for future work. By combining our encoding with ideas from syntax-guided synthesis, it would be possible to encode the problem of constructing an envy-free cake-cutting protocol with a bounded number of slices as a QBF instance, although we do not expect this approach to be fast enough to be practical. It may also be worthwhile to attempt an encoding of envy-freeness as a MILP instance; we suspect this may be faster than both the integer arithmetic used in *Crumbs* and the quantified real arithmetic used in *Slice*.

References

1. Audemard, G., Lecoutre, C., Lonca, E.: Proceedings of the 2022 XCSP3 competition. CoRR **abs/2209.00917** (2022). https://doi.org/10.48550/arXiv.2209.00917, https://doi.org/10.48550/arXiv.2209.00917
2. Aumann, Y., Lang, J., Procaccia, A.D.: Fair division (dagstuhl seminar 16232). Dagstuhl Reports **6**(6), 10–25 (2016). https://doi.org/10.4230/DagRep.6.6.10, https://doi.org/10.4230/DagRep.6.6.10
3. Aziz, H., Mackenzie, S.: A discrete and bounded envy-free cake cutting protocol for any number of agents. In: Dinur, I. (ed.) IEEE 57th Annual Symposium on Foundations of Computer Science, FOCS 2016, 9-11 October 2016, Hyatt Regency, New Brunswick, New Jersey, USA. pp. 416–427. IEEE Computer Society (2016). https://doi.org/10.1109/FOCS.2016.52, https://doi.org/10.1109/FOCS.2016.52
4. Aziz, H., Mackenzie, S.: A bounded and envy-free cake cutting algorithm. Commun. ACM **63**(4), 119–126 (2020). https://doi.org/10.1145/3382129, https://doi.org/10.1145/3382129

5. Bertram, N., Levinson, A., Hsu, J.: Cutting the cake: A language for fair division. CoRR **abs/2304.04642** (2023). https://doi.org/10.48550/arXiv.2304.04642, https://doi.org/10.48550/arXiv.2304.04642
6. Biere, A., Fleury, M.: Gimsatul, IsaSAT and Kissat entering the SAT Competition 2022. In: Balyo, T., Heule, M., Iser, M., Järvisalo, M., Suda, M. (eds.) Proc. of SAT Competition 2022 – Solver and Benchmark Descriptions. Department of Computer Science Series of Publications B, vol. B-2022-1, pp. 10–11. University of Helsinki (2022), http://hdl.handle.net/10138/359079
7. Brams, S.J., Pruhs, K., Woeginger, G.J. (eds.): Fair Division, 24.06. - 29.06.2007, Dagstuhl Seminar Proceedings, vol. 07261. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany (2007), http://drops.dagstuhl.de/portals/07261/
8. Clarke, E.M., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings. Lecture Notes in Computer Science, vol. 2988, pp. 168–176. Springer (2004). https://doi.org/10.1007/978-3-540-24730-2_15, https://doi.org/10.1007/978-3-540-24730-2_15
9. Corzilius, F., Kremer, G., Junges, S., Schupp, S., Ábrahám, E.: SMT-RAT: an open source C++ toolbox for strategic and parallel SMT solving. In: Heule, M., Weaver, S.A. (eds.) Theory and Applications of Satisfiability Testing - SAT 2015 - 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings. Lecture Notes in Computer Science, vol. 9340, pp. 360–368. Springer (2015). https://doi.org/10.1007/978-3-319-24318-4_26, https://doi.org/10.1007/978-3-319-24318-4_26
10. Klarreich, E.: How to cut cake fairly and finally eat it too. Quanta Magazine (October 2016), https://www.quantamagazine.org/new-algorithm-solves-cake-cutting-problem-20161006/
11. Kroening, D., Tautschnig, M.: CBMC - C bounded model checker - (competition contribution). In: Ábrahám, E., Havelund, K. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8413, pp. 389–391. Springer (2014). https://doi.org/10.1007/978-3-642-54862-8_26, https://doi.org/10.1007/978-3-642-54862-8_26
12. Lester, M.M.: Coptic: Constraint programming translated into C. In: Sankaranarayanan, S., Sharygina, N. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Paris, France, April 22-27, 2023, Proceedings, Part II. Lecture Notes in Computer Science, vol. 13994, pp. 173–191. Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_13, https://doi.org/10.1007/978-3-031-30820-8_13
13. Lester, M.M.: Crumbs: A DSL for Verifying Envy-Free Cake-Cutting Protocols using Bounded Integer Arithmetic (Jan 2024). https://doi.org/10.5281/zenodo.10205142, https://doi.org/10.5281/zenodo.10205142

14. Manthey, N.: Solving summle.net with SAT. In: Balyo, T., Heule, M., Iser, M., Järvisalo, M., Suda, M. (eds.) Proc. of SAT Competition 2022 – Solver and Benchmark Descriptions. Department of Computer Science Report Series B, vol. B-2022-1, pp. 70–71. University of Helsinki (2022), http://hdl.handle.net/10138/359079
15. de Moura, L.M., Bjørner, N.S.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings. Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24, https://doi.org/10.1007/978-3-540-78800-3_24
16. Procaccia, A.D.: Cake cutting: not just child's play. Commun. ACM **56**(7), 78–87 (2013). https://doi.org/10.1145/2483852.2483870, https://doi.org/10.1145/2483852.2483870