Clipping: A Semantics-Directed Syntactic Approximation

Dan R. Ghica
School of Computer Science
University of Birmingham, U.K.

Adam Bakewell
School of Computer Science
University of Birmingham, U.K.

Abstract

In this paper we introduce "clipping," a new method of syntactic approximation which is motivated by and works in conjunction with a sound and decidable denotational model for a given programming language. Like slicing, clipping reduces the size of the source code in preparation for automatic verification; but unlike slicing it is an imprecise but computationally inexpensive algorithm which does not require a whole-program analysis. The technique of clipping can be framed into an iterated refinement cycle to arbitrarily improve its precision. We first present this rather simple idea intuitively with some examples, then work out the technical details in the case of an Algol-like programming language and a decidable approximation of its game-semantic model inspired by Hankin and Malacaria's "lax functor" approach. We conclude by presenting an experimental model checking tool based on these ideas and some toy programs.

1. Introduction

Automatic software verification is well on its way to becoming an essential part of industrial software development. The success of verification toolkits such as SLAM [6] and BLAST [14] confirms the effectiveness of techniques for handling models with very large state spaces, such as abstraction-refinement and on-the-fly model construction. However, it is fair to say that the success of fully automated software model checking (SMC) has been restricted to small, compact, self-contained programs such as device drivers. Scaling this technique up to large and very large programs has remained an open question. The main technique employed relies on using slicing [20] to syntactically reduce the size of the program only to those components that are relevant to a property to be checked [9].

One common denominator of the conventional solutions mentioned above is an emphasis on static analyses of a rather syntactical nature, combined with state-exploration or predicate-transformer operational semantics of the programming language. In a series of papers [1, 13, 10, 11, 5], the first author and collaborators have argued for an alternative approach, based on a compositional model constructed denotationally using game semantics [15, 2].

In our papers we stressed the usefulness of the "external" aspect of compositional verification: we can prove the correctness of a program by proving the correctness of its components. We argued that a denotational-semantic approach, which has compositionality built-in, is the most simple and technically elegant way of achieving it. Our previous work focused on adapting the conventional state-space saving heuristics to the game-semantic setting, and building a state-of-the-art games-based SMC tool[1]. In doing that we often exploited the "internal" compositionality of the game model, i.e. the fact that the model of a term is constructed as a composition of its sub-terms models. The technique in this paper combines and exploits for the first time both the "external" and the "internal" compositionality of the game-semantic model into an innovative abstraction algorithm that, we argue, can be an important step towards "scaling up" automatic verification to large software projects.

The basic, and rather simple, idea of *clipping* is best understood in contrast with *slicing*. Slicing [20] is a syntactic under-approximation which is calculated to be sound for a fixed set of program properties. It removes parts of a program that are considered irrelevant, basically by replacing them with constants in the abstract syntax tree of the program; by contrast, clipping will replace parts of the program that are considered (potentially) irrelevant with fresh identifiers. This is where the magic of denotational approach kicks in: the denotational model of a fresh identifier of any given type is defined as a universal quantifier over all possible behaviours of terms of a given type; it is *the most general behaviour* at that type. Consequently, the identifier is an over-approximation

[1] `www.cs.bham.ac.uk/research/projects/mage`.

for any given concrete sub-term it replaces! At the same time, the semantic model of the free identifiers is of fixed size, possibly much smaller than the models of the sub-terms they replace; in extreme cases whole programs can be replaced by one single identifier.

Whereas slicing requires a whole-program analysis of the source code, clipping can work locally on a piece of code, which we call *the target*, and its immediate syntactic context. Also, slicing has a substantial computational cost (the more precise the analysis the more expensive), but clipping is computationally trivial. Finally, slicing produces a fixed and precise approximation, while clipping produces an imprecise approximation, that can be however improved to any given degree through iterated refinement. Because it is computationally cheap and flexible, clipping could be a successful method for large programs.

Consider the example in Fig. 1, contrasting a program with an inter-procedural slice [19] and with a clipped term of similar shape. Slicing is an under-approximation which replaces selected commands in the program with the null command (lines 4, 7, 10) whereas clipping is an over-approximation that replaces selected commands (or, more generally, selected branches in the syntax tree) with free identifiers (X0, X1, X2, X3) representing undefined functions of an appropriate type called *stumps*; the stumps take as arguments the free identifiers collected from the clipped branch.

Clipping can be refined in the usual way: the counterexamples found in the approximated program are potentially spurious, but they can be used to indicate what parts of the program may trigger errors. The refinement then simply consists in replacing a clipping identifier (a stump) with a more precise term, consisting of a sub-term that has been itself suitably clipped. We will show that in general the clipping-refinement algorithm is sound and complete.

A methodological point. Virtually no program is self contained. Even if we want to verify one single source file, the code will make function calls to some libraries; if the source file does not contain the `main()` function, then the code there is only executed insofar as it is called from other functions. In order to verify the code in one single source file, one needs to find a way to manually "close" the source code syntactically by adding various function stubs and harnesses or by including the actual source code of library routines, in case they are executable. So, even though the actual *verification* is automatic, the preparation of the source code can be laborious.

A further down-side of the manual closure of source code into an executable is that some assumptions must be made regarding the way a library routine will be used. Because real software is rarely specified, much less so formally, a bug is almost always a result of the incorrect interaction between the caller and the callee. It is sometimes difficult to decide whether a bug is not actually a feature! If a routine is tested in the context of a hand-crafted execution harness and a bug is found (or not) it is arguable whether that is because of the routine or the incorrectness (incompleteness, respectively) of the harness.

Clipping and its iterated refinement are meant to tackle exactly this problem, by checking the error-free-ness of a target sub-term in a program initially in a most-general context generated by the semantics, which is gradually refined with actual contexts extracted from the program. In this way we can say with certainty that a bug is caused by an actual interaction existing in the program.

2. A higher-order imperative language

We illustrate clipping and refinement of clipping as applied to an Algol-like language with `abort`, a simplified and recursion-free version of Algol [3] with local exceptions [16]. It is a highly expressive programming language combining higher-order procedures with local state, a language which has been the subject of extensive semantic analysis. We use this language in order to give a focused presentation, but the algorithm works in general for any language with a sound denotational model in which models of terms can be soundly and decidably approximated.

The base types are commands (`com`), (finite) integers (`int`), assignable variables (`var`). The function types are of the form $T \to T'$, where T, T' are types.

The constants of the language are `skip` (the empty command) and integer constants. The operations of the language are arithmetic, branching, iteration, assignment, dereferencing, sequential composition and local variable declaration. Finally, the language has a countable set of failure labels S so that $\texttt{abort}(s) : \texttt{com}$, $s \in S$. Terms are formed from the above constants and combinators and the simply-typed lambda calculus (variables, abstraction, application). We denote well-typed terms M with free identifiers in Γ and type T as $\Gamma \vdash M : T$.

We will use the functional-style syntax or a sugared more conventional syntax, depending on whichever is more convenient. The two are equivalent, e.g. $\texttt{int}\ x;\ x := x + 1$; vs. $\texttt{new}(\lambda \texttt{x.asg x}(\texttt{add}(\texttt{der x})1))$.

In this language it is straightforward to encode arrays (as tuples) and assertions. This is a simplified

```
1      let add(x,y)=          let add(x,y)=          let add(x,y)=
2          x:=x+y;                x:=x+y;                X3(x, y);
3      int sum, i;            int sum, i;            int sum, i;
4      sum := 0;                                     X0(sum);
5      i := 1                 i := 1                 i := 1;
6      while i<11{            while i<11{            while i<11{
7        add(sum, i);
8        add(i, 1);}              add(i, 1)}             X1(add,sum,i)}
9      print sum;                                    X2(sum);
10     print i                print i                print i
```

Figure 1. Original code vs. slicing vs. clipping

version of Idealized Algol with local exceptions [16], in that the `catch` construct is not provided. We call this language IAf (Idealized Algol with failures) [10]. In the paper we assume some familiarity with basic game-semantic concepts

In the following we are interested in *safety* properties, i.e. ensuring that a program cannot perform a certain undesirable action.

Definition 1 *Given a subset of failure Player moves S we say a term is S-*safe *if no S-move occurs in $[\![\Gamma \vdash M : T]\!]$.*

In [10] we give an operational interpretation of S-safety: any program $\mathcal{C}[M]$ can fail only if the context $\mathcal{C}[-]$ fails, i.e. the term is operationally safe in any syntactically safe contexts.

3. Context closure using game semantics

As mentioned, game semantics is useful because it provides a way to model open terms, terms in which free identifiers occur. The game model for a free identifier includes all the behaviours that terms in the program of that type can have. Therefore, the game model provides a *closure* to the context by creating *the most general environment* in which a term can operate. This is akin to the context closure in [8], but generalized to program contexts.

We can give strategies $\sigma : A \Rightarrow B$ concrete representations by Moore-style finite state machines with output alphabet its set of moves $M_A + M_B$, empty input alphabet, and transition function δ_σ. Consider for example the program in Fig. 2. We refer to the framed sub-term as the "calling context" of `push`. The game model for this term can be represented by the transition system in Fig. 3. Opponent failure moves, representing errors caused by the context, are not interesting and we omit them from the representation.

```
test : int->com |-
int n := 0, s[3];
int push(int x) {assert(n < 3);
                 s[n] := x; n++; !x}
test(push 0)
```

Figure 2. Stack push source code, in context

Since IAf is a sequential language, the only way the free identifier `p` can use its argument is to call it in an arbitrary sequence. So all the `test` function can do is push 0 on the stack an arbitrary number of times. If `test` calls its argument 3 times or fewer normal termination occurs (move *a*); if `test` calls its argument 4 times the assertion fails and the program aborts.

For higher-order terms, the models for free identifiers are more complex. It is therefore impractical to use the game models *per se* to semantically close the environment for verification. We will trade off precision for simplicity of representation, similarly to Hankin and Malacaria's "lax functor" for approximating game models [17], by dropping the justification pointers and relaxing some of the combinatorial constraints that lead to non-finite state representations. Given the usual concept of an *arena*, where M is its set of *moves* and λ a function assigning Proponent/Opponent and Question/Answer polarities to each move.

Definition 2 *Given an arena $A = \langle M, \vdash, \lambda \rangle$ we define the*

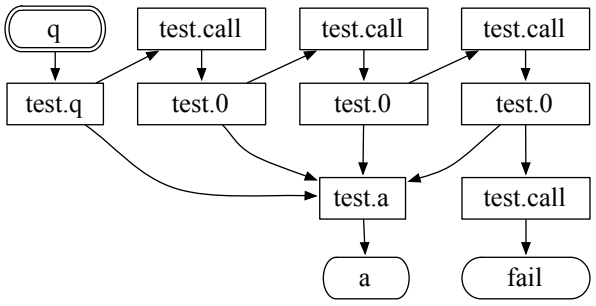


Figure 3. Model for `test(push 0)`

language of approximate plays *as the language $\mathcal{L}(-)$ accepted by the state machine $\hat{A}$ with set of states M and transition relation $\delta(m,n)$ if $\pi_1 \circ \lambda(m) \neq \pi_1 \circ \lambda(n)$ and $n \notin I_A$ and m not an answer to any I_A.*

In words, we create a finite automaton with the enabling relation as transitions, augmented with new transitions from each P-move to each O-move, except the initial question and its answers.

Definition 3 *Given an arena A we define the* approximate identity *$\widehat{id}_A : A \Rightarrow A$ as the language $\kappa(\mathcal{L}(\hat{A}))$ where $\kappa = \begin{cases} in_r(m)in_l(m) & \text{if } \pi_1 \circ \lambda(m) = O \\ in_l(m)in_r(m) & \text{if } \pi_1 \circ \lambda(m) = P, \end{cases}$, where in_l, in_r are the two injections in the co-product $A + A$.*

The approximate identity defined above is an approximation in the following sense: let $\overline{\sigma}$ be a strategy with all justification pointers removed.

Proposition 4 *For any arena A, $\widehat{id}_A \supseteq \overline{id}_A$.*

Proof: The set of approximate plays on an arena $\mathcal{L}(\hat{A})$ is the set of all alternating plays, therefore it includes the set of all legal plays. The approximate identity $\widehat{id}_A$ is the copycat behaviour on the (larger) set $\mathcal{L}(\hat{A})$. □

The approximate identity, however is precise at ground and first order types.

Proposition 5 *For any ground type or first-order arena A, $\widehat{id}_A = \overline{id}_A \cong id_A$.*

The isomorphism is the unique reconstruction of the justification pointers.

It follows also immediately from definitions that

Proposition 6 *For any arena A, $\widehat{id}_A$ is a regular language.*

Approximate identities can be used to close an environment with behaviour that is richer than what is actually possible to realize in IAf. As an approximation technique, this is acceptable loss of precision.

For simplicity we can (and will) actually use just the language of approximate plays as the approximation of identity. The copy-cat move replication is only necessary in order to provide the right hook-ups for application. If we are only modelling normal-form terms, and are interested in safety properties we can use the simpler set of approximate plays, which is in fact isomorphic to the approximate identity through κ.

The operation of composition must also be approximated. If the strategy for $\sigma : A \to B$ is decomposed linearly $\sigma : !A \multimap B$ then the composition of two strategies $A \xrightarrow{\sigma} B \xrightarrow{\tau} C$ requires the promotion of strategy from $\sigma : !A \multimap B$ to $\sigma^\dagger : !A \multimap !B$. This decomposition is rather standard in game semantics [3].

Algorithmically, promotion corresponds to an interleaving of arbitrarily many plays from σ with themselves (subject to legal-play constraints), and it does not have a finitary model in general.[2]

Definition 7 *Given finite-state machines representing strategies $\sigma : A \Rightarrow B$ and $\tau : B \Rightarrow C$ we define the finite-state machine $\tau \bullet \sigma$ over alphabet $M_A + M_B + M_B + M_C$ with transition relation given by $(m,n) \in \delta_{\tau\bullet\sigma}$ if and only if*

1. $m, n \notin \mathrm{in}_2(M_B) \cup \mathrm{in}_3(M_B) \wedge (m,n) \in \delta_\tau + \delta_\sigma$
2. $m, n \in \mathrm{in}_2(M_B) \wedge (m,n) \in \delta_\sigma \wedge \lambda_B(m) = O$
3. $m, n \in \mathrm{in}_3(M_B) \wedge (m,n) \in \delta_\tau \wedge \lambda_B(m) = P$
4. $\exists p \in M_B . m = \mathrm{in}_2(p) \wedge n = \mathrm{in}_3(p) \wedge \lambda_B(p) = P$
5. $\exists p \in M_B . n = \mathrm{in}_2(p) \wedge m = \mathrm{in}_3(p) \wedge \lambda_B(p) = O.$

The FSM defined above is an approximation of the interaction sequence of the two stategies, constructed by replacing P-to-O transitions within the two B arenas with transitions between the arenas. Note that the new added transitions are the "interaction links" mentioned in [17].

Given arena A we define the "restriction" function out_A on sets of sequences over $M \supseteq M_A$ that removes all moves not in M_A. This is an approximation in the same sense as Prop. 4:

Proposition 8 *For any strategies $\sigma : A \Rightarrow B$ and $\tau : B \Rightarrow C$, $\overline{\tau \circ \sigma} \subseteq out_{A+C}(\tau \bullet \sigma)$.*

Proof: The proof relies on the visibility condition which leads to a standard "switching condition" for strategy composition: moves from A and C cannot occur consecutively; at least one move from B must separate them. □

In Fig. 4 we show the approximate composition for the stack in Fig. 2, but now with a higher-order calling context `test(push)` rather than `test(push 0)` (we only show the fail move associated with the assertion). On the left is the state-machine for `push` and on the right the approximation of `test`, the identity on $(\mathtt{int} \Rightarrow \mathtt{int}) \Rightarrow \mathtt{com}$. The following transitions are represented: solid arrows are original transitions, dotted arrows are from the approximation of identity, dashed arrows are interaction links for composition, and the grey arrows are removed during composition. This composition contains traces that correspond to a serial (as in `push 7; push 9`) and nested

[2] The approximation used in [17] applies only to control-flow analysis. Our approximation is significantly more precise while remaining finite.

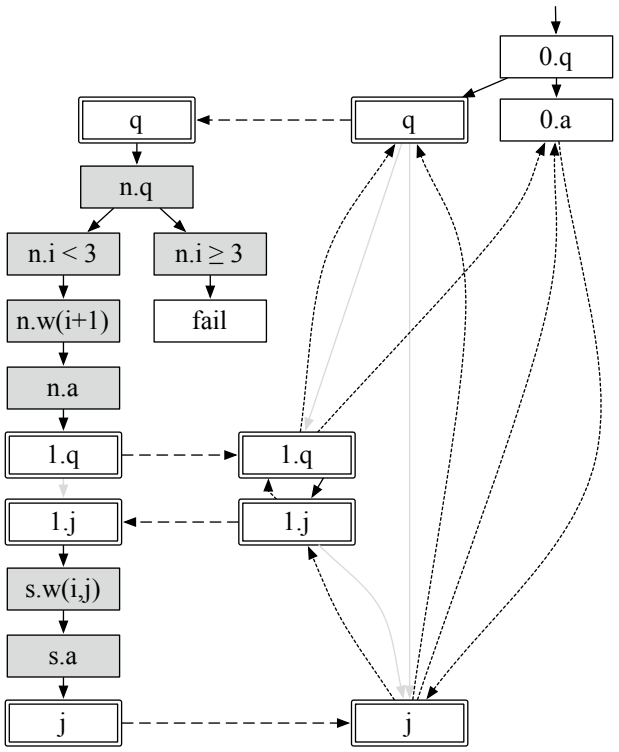


Figure 4. Approximation `push` • $\widehat{\texttt{test}}$

(as in `push(push(9))`) uses of the argument but also traces that cannot be realised by any context, for example $0.q \cdot q \cdot n.q \cdot n.0 \cdot n.w(1) \cdot n.a \cdot 1.q \cdot q \cdot n.q \cdot n.1 \cdot n.w(2) \cdot n.a \cdot 1.q \cdot 0.a$. Such traces correspond to a context that interrupts the evaluation of `push` when it asks for an argument, and cannot be defined in IAf.

A precise approximation should have only traces of the first and second kind, but such a language is not finite state [18].

Definition 9 *Let the* approximate semantics $(\!|M|\!)$ *be:*

$$\begin{aligned}
(\!|\Gamma \vdash \mathbf{k}|\!) &= \overline{[\![\Gamma \vdash \mathbf{k}]\!]} \\
(\!|x : T \vdash x : T|\!) &= \widehat{id}_T \\
(\!|\Gamma \vdash \lambda x.M : T \to T'|\!) &= \Lambda\big((\!|\Gamma, x : T \vdash M : T'|\!)\big) \\
(\!|\Gamma \vdash MN : T'|\!) &= out_{\Gamma,T'}\big((\!|\Gamma \vdash M : T \to T'|\!) \\
&\qquad \bullet\, (\!|\Gamma \vdash N : T|\!)\big),
\end{aligned}$$

where Λ is the currying isomorphism in the category of games and $\mathbf{k}$ is any IAf constant or combinator.

The method of approximation presented here, akin to Hankin and Malacaria's "lax functor," is both effective and sound.

Theorem 10 *For any IAf term* $\Gamma \vdash M : T$, $\overline{[\![\Gamma \vdash M : T]\!]} \subseteq (\!|\Gamma \vdash M : T|\!)$. *Moreover, the set* $(\!|\Gamma \vdash M : T|\!)$ *is regular.*

Proof: Immediately by induction on syntax. All approximations of constants and free identifiers are finite-state and sound (Prop. 4). Composition is sound and preserves finite-state-ness (Prop. 8). □

Finally, note that

Lemma 11 *If* Γ *has only ground and first-order identifiers and* M *is ground type or first order in beta-normal form then* $[\![\Gamma \vdash M : T]\!] = (\!|\Gamma \vdash M : T|\!)$.

Proof: This is the main result of [12]. Since no higher-order identifiers or applications occur inside the term then the approximations of identity and composition are not used. For language constants, the approximate semantics is the same as the precise semantics. □

4. Clipping and iterated refinement

Using the approximation techniques explained in the previous section we can abstract the context of some target sub-term by replacing, in the syntax tree, parts of the concrete context of the sub-term with free identifiers. As a running example we consider the same `push` procedure from Fig. 2, but in a concrete calling context: `push 0; push 0; push 0`.

Example 1 The syntax tree of the concrete program is given in Fig. 5(a).

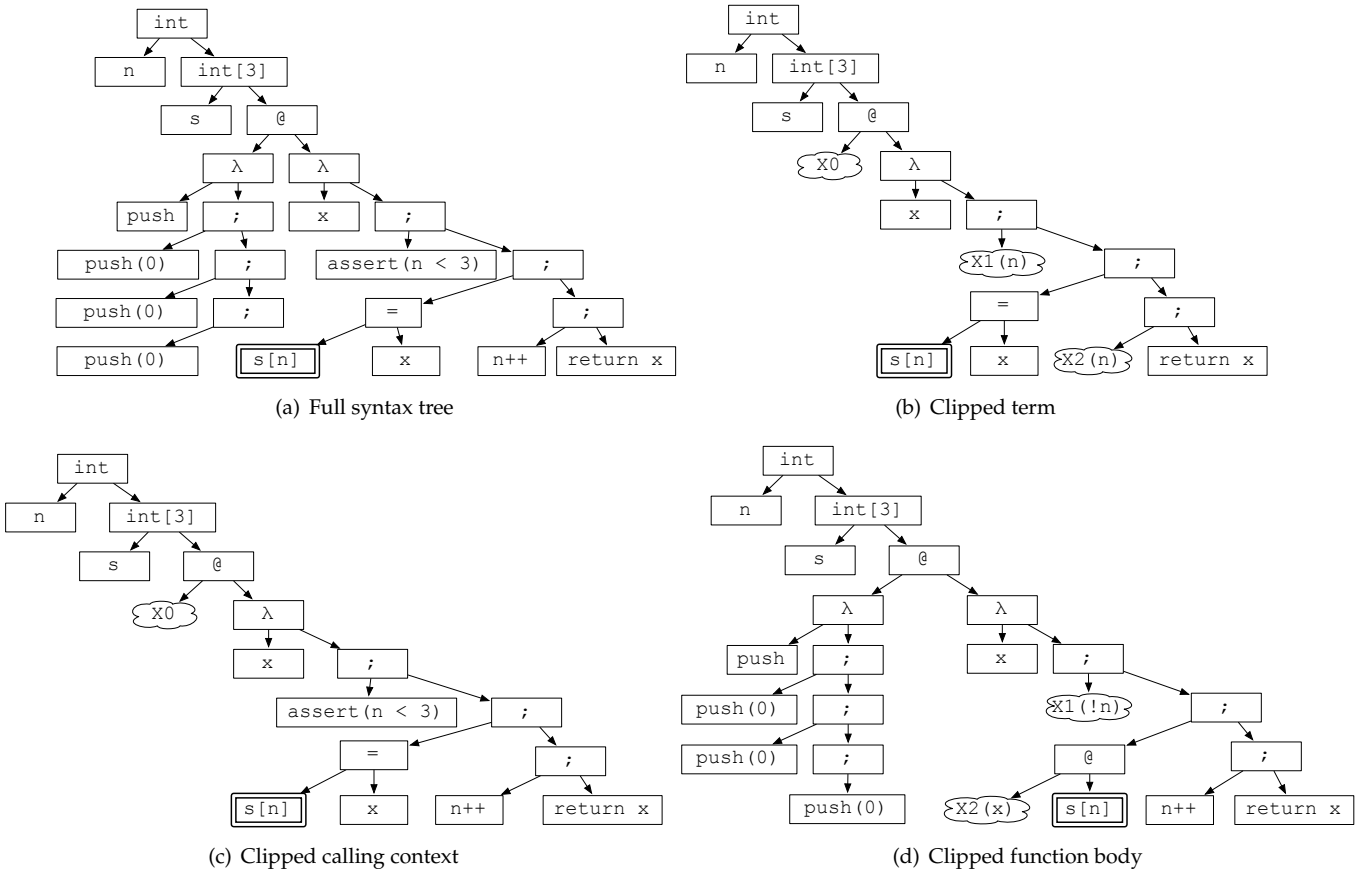
Suppose that we set the target to be the sub-term `s[n]`. The only possible failure in this target is on the array access operation and we assume this particular failure has a distinguished label. In the context of this set of examples by safe (unsafe) we mean the absence (presence) of the move associated with this particular failure. A program failing with a different failure label will be considered safe. The coarsest clipping is obtained by following the path from the root to the term in the syntax tree and removing all other branches except local-variable binders, and replacing them with free identifiers. The result is the syntax tree in Fig. 5(b), which represents the term

$$\texttt{int n; int s[3]; X0}(\lambda\texttt{x.X1(n); s[n]} = \texttt{x; X2(n); !x}),$$

which upon model-checking is *unsafe*: the procedure `X2` can assign any value to n, and can cause an access error. This is, of course, a very rough approximation.

Example 2 Another clipped version of the same term is given in Fig. 5(c). In this instance, only the calling context has been abstracted (by `X0`) while the body of `push` is fully refined. The term is

$$\texttt{int n; int[3] s;}$$



(a) Full syntax tree

(b) Clipped term

(c) Clipped calling context

(d) Clipped function body

Figure 5. Examples of clipping

$$\texttt{X0}(\lambda \texttt{x.assert}(\texttt{n} < 3);\ \texttt{s}[\texttt{n}] = \texttt{x};\ \texttt{n}{+}{+};\ !\texttt{x})$$

In this instance we can model-check the term and see that the target is *safe*. The abstracted calling context `X0` can call the body of `push` an arbitrary number of times, including nested calls and non-realizable calls, but any call that results in a value of `n` larger than `3` will cause the assertion to fail and stop the flow of execution before reaching the target. Since the assertion is not part of the target its failures do not count as target failures.

In this instance therefore, the target is proved correct without instantiating the calling context.

Example 3 In another possible clipped context for the same target we include the concrete calling context but we partly abstract the body of the `push` procedure, as in Fig. 5(d). The corresponding term is

```
int n; int[3] s;
int push(x){X1(!n); X2(x, !s[n]); n++; return !x)}
push(0); push(0); push(0)
```

In this instance too the target can be proved safe. Even though some of the function body is missing, the only relevant computations for the safety of the target are the increment of the index and the array access. Given that the concrete calling context only has three `push` statements, no overflow will occur and the term can be proved safe.

These simple examples illustrate the flexibility of clipping and its ability to remove irrelevant detail both from the function body and from the calling context in order to reduce the size of the program being checked. Clipping can be refined in the usual counterexample-guided fashion. Consider for example the clipping in Example 1. The shortest possible play that ends with a failure in the target will indicate calls to `X0`, `X1` and `X2`. So these identifiers are natural targets for refinement by replacing them with a more concrete context. Refining $X1$ and $X2$ leads to the less abstract context in Example 2, which can prove the target to be correct.

Given two terms we write $M \in N$ to indicate that M is a sub-term of N. Given a fixed target sub-term C

$$\Gamma \vdash M : T \trianglelefteq_C \Gamma \vdash M : T \tag{1}$$

$$x_0 : T_0, \ldots, x_n : T : n \vdash M : T \quad \trianglelefteq_C \; x_0 : T_0, \ldots, x_n : T_n, X : T_0 \to \cdots \to T_n \to T \vdash X x_0 \cdots x_n : T, \qquad C \notin M, X \text{ fresh} \tag{2}$$

$$\frac{\Gamma \vdash F : T \to T' \trianglelefteq_C \Gamma, \Delta \vdash \widetilde{F} : T \to T' \qquad \Gamma \vdash M : T \trianglelefteq_C \Gamma, \Delta \vdash \widetilde{M} : T}{\Gamma \vdash FM : T' \trianglelefteq_C \Gamma, \Delta \vdash \widetilde{F}\widetilde{M} : T'} \tag{3}$$

$$\frac{\Gamma, x : T \vdash M : T' \trianglelefteq_C \Gamma, \Delta, x : T \vdash \widetilde{M} : T'}{\Gamma \vdash \lambda x : T.M : T \to T' \trianglelefteq_C \Gamma, \Delta \vdash \lambda x : T.\widetilde{M} : T \to T'} \tag{4}$$

$$\frac{\Gamma \vdash M : T' \trianglelefteq_C \Gamma, \Delta \vdash \widetilde{M} : T'}{\Gamma, x : T \vdash M : T' \trianglelefteq_C \Gamma, \Delta, x : T \vdash \widetilde{M} : T'} \tag{5}$$

Figure 6. Definition of the clipping relation

we define the clipping relation $\trianglelefteq_C$ inductively on type judgements of IAf as in Fig. 6. Rule 1 indicates that any term can be (trivially) clipped into itself. Rule 2 is the rule that actually performs a clip by replacing a term (which does not contain the target) by a fresh identifier (*stump*) applied to its free identifiers. Rules 3 and 4 propagate clipping through the syntax tree, and rule 5 is a weakening operation that allows the removal of unused identifiers.

Lemma 12 *For any set of moves S, $\Gamma \vdash M : T$ is S-safe only if for any $C \in M$ and any $\Gamma, \Delta \vdash \widetilde{M} : T$, such that $\Gamma \vdash M : T \trianglelefteq_C \Gamma, \Delta \vdash \widetilde{M} : T$, is safe.*

Proof: Immediate by induction on the syntax of the term-in-context $\mathcal{C}[M]$. The only non-trivial case, Rule 2 in Figure 6 comes directly from the semantic definition, because the copy-cat strategy defining stump X has more plays than any other term at that type. □

Let us assume, with no loss of generality, that the failure labels in a selected target are distinct. The general counter-example guided clipping refinement (CEGAR) algorithm, in the style of [7], for the S-safety of a target M in context $\mathcal{C}[-]$ is given in Fig. 7.

Theorem 13 *For any target term M and ground-type context $\mathcal{C}[-]$:*

1. *if the CEGAR algorithm reports SAFE then the program $\mathcal{C}[M]$ is S-safe.*
2. *if the CEGAR algorithm reports UNSAFE then the program $\mathcal{C}[M]$ is S-unsafe.*
3. *the CEGAR algorithm terminates.*

Proof: It is immediate by structural induction on the derivation of $\mathcal{C}[M]$ that all terms $\widetilde{\mathcal{C}[M]}$ produced in steps 7 to 11 are proper clippings. Then:

1. from Lem. 12 and Thm. 10 (part 1).
2. from the fact that the failure traces without occurrences of X moves from any $(\!|\widetilde{\mathcal{C}[M]}|\!)$ will also occur in $(\!|\mathcal{C}[M]|\!)$. Since we reduce the term to normal form before model-checking, $(\!|\mathcal{C}[M]|\!) = [\![\mathcal{C}[M]]\!]$ (Lem. 11), so the failure trace occurs in the concrete model.
3. Step 2 terminates because we do not have recursion. Thm. 10 (part 2, which guarantees that Step 3 terminates) together with the fact that the syntactic refinement chain is bounded by the concrete syntax, and each step creates a strictly more refined version of the context guarantee overall termination. □

The refinement step is sketched out diagrammatically in Fig. 8.

The CEGAR algorithm is generic, in the sense that the selection of the failure trace $p \cdot m$ and refinement term M' for X are non-deterministic. The correctness and termination of the algorithm are independent of such choices. However, the efficiency of the algorithm demands that these choices are informed by certain heuristics which we shall discuss in the implementation section. Another important optimisation that we must consider is the stopping criterion for UNSAFE (Step 5). As is, the algorithm stops only when it discovers a "concrete" error trace. As discussed in [5], an early detection of counterexample traces that include abstracted entities can greatly improve the effectiveness of the algorithm.

5. Examples

This section illustrates the impact of clipping on some examples and discusses key technicalities that

1. *Calculate an arbitrary initial clipped term* $\widetilde{M}$.
2. *Calculate the beta-normal form* M_β *of* $\widetilde{M}$
3. *Model-check* $(\!|M_\beta|\!)$ *for* S*-safety.*
4. *If* M_β *is* S*-safe then report SAFE and stop.*
5. *If there are no moves for stumps* X *in* $(\!|M_\beta|\!)$ *then report UNSAFE and stop.*
6. *Otherwise take a failure trace* $p \cdot m \in (\!|M_\beta|\!)$ *where* $m \in S$.
7. *Take a move in* p *for some stump* X.
8. *Identify sub-term* M_X *of* $\mathcal{C}$ *abstracted by* X.
9. *Identify an arbitrary target sub-term* M_0 *of* M_X.
10. *Let* $\widetilde{M}_X$ *be the clipping of* M_X *relative to* M_0.
11. *Let* $\widetilde{M}$ *now be the context obtained by replacing* X *with* $\widetilde{M}_X$ *in* M.
12. *repeat from Step 2*

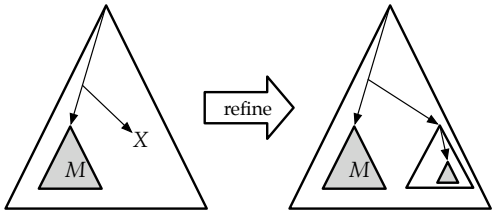
Figure 7. CEGAR for clipping

arise with clipping and refinement in practice.

The term below defines a memoisation table module. The `user` may apply `f` any number of times; `f` returns `0` if it has not seen the argument previously. If `n` is always in `0` to `Z-1` `f` then returns `1` if it has seen `n`. The model of `bigmemo` contains order 2^Z traces so its elimination by clipping leads to a substantial reduction in model size.

```
T bigmemo(user:(int -> int) -> T){
  var[Z] m; var i := 0;
  while i < Z do {m[i] := 0; i := i + 1};
  var f(int n){
    var saw := m[n % Z];
    m[n % Z] := 1; saw};
  user(f)}
```

Figure 8. Refinement step



Z	M	no clip	clip	clip under	clip opt
2	2	0.6	0.1	0.1	0.1
2	3	4.8	0.5	0.3	0.2
2	4	41.3	2.5	1.6	0.5
2	5	292.3	13.4	7.9	2.2
2	6	2340.9	56.0	36.9	10.8
4	2	2.1	0.7	0.3	0.1
8	2	8.7	1.5	0.9	0.4
16	2	36.2	16.1	12.2	4.3
32	2	173.3	34.8	26.7	8.1

Table 1. Model checking times (seconds)

To set a benchmark we define the following cardinality application as a `bigmemo` user that counts the number of distinct letters in word `w`.

```
int card(var[M] w){
  bigmemo
    (fun f:int -> int .
     var c := 0;
     var i := 0;
     while i < M do{
       if f(w[i]) = 0 then c := c + 1;
       i := i + 1};
     c)
```

The *no clip* column of Table 1 shows the times taken by MAGE to model check the highlighted target for array access safety (i.e., that `w[i]` never goes out of bounds), for various table sizes (`Z`) and word sizes (`M`).

The following clipping contains just enough detail to verify the safety of highlighted target array access term `w[i]`:

```
int card(var[M] w){
  X0
    (fun f:int -> int .
     var c . X1(c);
     var i := 0;
     while i < M do{
       X2(f,w[i],c);
       i := i + 1};
     c)
```

Intuitively, the big advantage of this clipping should be that by eliminating the entire memo table with `X0`, and the details about what to do with counter `c`, should give a substantial reduction in model size. Indeed, the MAGE verification times as shown by the *clip* column of Table 1 show a significant improvement.

For iterated refinement, we start with an arbitrarily clipped term:

```
int card(var[M] w){
  X0
    (fun f:int -> int .
     var c,i .
     X1(c,i);
     while X2(i) do{
       if f(w[i])=X3() then X4(c);
       X5(i)};
     c)
```

Following the refinement algorithm of Figure 7 we can reintroduce parts of the terms abstracted by `X0`, `X1` or `X2`. Note that `X0` would be a very poor choice, because putting the memo table back reintroduces a lot of state and therefore most of the model.

The stumps that modify `i` are much better choices because providing more information about `i` is likely to eliminate the error if it is spurious. Following this rule we can refine

1. `X1(c,i)` into `X1(c); i := 0`
2. `X2(i)` into `i < M`
3. `X5(i)` into `i := i + 1`.

This refinement gives the safe clipping used earlier in this section.

5.1. Heuristics

The standard stump models are over-approximations, but we can also use semantically-motivated sound under-approximations to further reduce the size of the model. For example we can omit occurrences of any assignable variable v as an argument to the stump if all occurrences of the variable in the abstracted sub-term are dereferenced. Another simple and safe under-approximation which can reduce the size of the model is dropping occurrences of non-locally defined functions used as arguments to stumps. Non-locally defined functions cannot have any side-effects that impact local state, and their interaction with the other arguments can be simulated directly by the behaviour of the stump itself. For example, in the `card` clipping we can omit `f` from stump `X2`.

```
int card(var[M] w){
  X0
    (fun f:int -> int .
     var c . X1(c);
     var i := 0;
     while i < M do{
       X2(w[i],c);
       i := i + 1};
     c)
```

The *clip under* column of Table 1 shows the impact of this change.

Furthermore, when the clipped branches of the syntax tree are small, applying clipping can actually be counterproductive because the stump model can be larger than the model of the sub-term it approximates. Intuitively, clipping takes a subprogram with a particular model complexity and replaces it with `X(v1,..,vn)` which has a model that says "after execution the state is the same except these `n` variables could have changed to any value in their domain". In any situation where this *model* transformation always leads to worse performance of the checking tool the clipping transformation should be avoided.

Returning to the card example, the clipped sub-expressions that assign to `c` are simple instances of this problem: `X1` replaces the assignment of a constant (a term with a constant-size model) with something that says `c` can become anything — a linear-size model. The following reformulation avoids these bad stumps with an improvement in verification time shown in column *clip opt* of Table 1.

```
int card(var[M] w){
  X0
    (fun f:int -> int .
     var c := 0;
     var i := 0;
     while i < M do{
       if X1(w[i]) then c := c + 1;
       i := i + 1};
     c)
```

In general it is often the case that abstracting sub-terms that are very small or that do not include local state declarations or loops may yield clipped terms that are more expensive than the concrete sub-term. Even for terms that are worth approximating, the model checker performance may be significantly improved by optimizing the stumps. The most obvious candidates are situations like sequences `X1(v1,v2); X2(v2,v3)` which has an effect on assignable variables identical to the model-smaller clipped term `X3(v1,v2,v3)`. Similarly, applications like `X1() X2(v)` which are equivalent to `X3(v)`, and `X1(X2(f))` is equivalent to `X3(f)`.

In a similar vein, refinements are only worthwhile if they reintroduce detail (meaning elimination of impossible sub-traces) to the model. Refinements that optimize back to the original are not useful! Reintroducing just a sequencing point or function application

is never helpful. With control structures the picture is more nuanced: refinements like `X0(v1,v2,v3)` to `if X1(v1) then X2(v2) else X3(v3)` are likely to help; but to `if X1(v1,v2) then X2(v3) else X3(v3)` are not.

A systematic treatment of such optimisations will be attempted in future work.

6. Related and further work

Clipping and its refinement form an algorithm of very low complexity; it is a constant time and constant memory overhead over the reachability test (Step 3) in Fig. 7 once the syntax tree of the term-in-context is created. The resulting approximations are coarse but they can eliminate arbitrarily large branches from the term to be model-checked, and the approximation can be refined iteratively. We think these features recommend our technique for usage on large code bases. It would be thus interesting to consider how this technique could apply to a language such as C. A simpler, more restricted version of clipping, called "structural abstraction" has been recently proposed for C [4]. Although it is very simple, basically just placing indeterminate values in all variables in the cone of influence of an abstracted function, experimental evidence as to its potential is convincing.

Clipping relies crucially on the existence of a decidable denotational-semantic model for the programming language in which the program to be verified is written, a model that can be obtained using *game semantics*. Although no common language (such as C) has a game model, many language features have been already modelled in the context of various idealized languages. We believe the game semantic models of common programming languages are within reach, and we will need such a model in order to test clipping on realistic examples.

Obtaining finitary approximations of higher-order identity and of composition is not automatic, and some models may be easier to approximate than others. This analysis will need to be carried-out in the context of each programming language. A finitary approximation of recursion is possible in a trivial way in our framework by iterated unfolding followed by clipping. However this will make the CEGAR algorithm semi-decidable; even for unsafe programs it is necessary to schedule the unfoldings of the recursion operators in the program fairly in order to prevent divergent local analyses.

Acknowledgement. This research was supported by UK EPSRC grants EP/D070880/1 and EP/D034906/1.

References

[1] S. Abramsky, D. R. Ghica, A. S. Murawski, and C.-H. L. Ong. Applying game semantics to compositional software modeling and verification. In *TACAS*, pages 421–435, 2004.

[2] S. Abramsky, R. Jagadeesan, and P. Malacaria. Full abstraction for PCF. *Inf. Comput.*, 163(2):409–470, 2000.

[3] S. Abramsky and G. McCusker. Linearity, sharing and state: a fully abstract game semantics for Idealized Algol with active expressions. *Electr. Notes Theor. Comput. Sci.*, 3, 1996.

[4] D. Babic and A. J. Hu. Structural abstraction of software verification conditions. In *CAV*, pages 366–378, 2007.

[5] A. Bakewell and D. R. Ghica. On-the-fly techniques for game-based software model checking. In *TACAS*, pages 78–92, 2008.

[6] T. Ball and S. K. Rajamani. The SLAM toolkit. In *CAV*, pages 260–264, 2001.

[7] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV*, pages 154–169, 2000.

[8] C. Colby, P. Godefroid, and L. J. Jagadeesan. Automatically closing open reactive programs. In *PLDI*, pages 345–357, 1998.

[9] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng. Bandera: extracting finite-state models from Java source code. In *ICSE*, pages 439–448, 2000.

[10] A. Dimovski, D. R. Ghica, and R. Lazic. Data-abstraction refinement: A game semantic approach. In *SAS*, pages 102–117, 2005.

[11] A. Dimovski, D. R. Ghica, and R. Lazic. A counterexample-guided refinement tool for open procedural programs. In *SPIN*, pages 288–292, 2006.

[12] D. R. Ghica and G. McCusker. The regular-language semantics of second-order Idealized Algol. *Theor. Comput. Sci.*, 309(1-3):469–502, 2003.

[13] D. R. Ghica and A. S. Murawski. Compositional model extraction for higher-order concurrent programs. In *TACAS*, pages 303–317, 2006.

[14] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with BLAST. In T. Ball and S. K. Rajamani, editors, *SPIN*, volume 2648 of *Lecture Notes in Computer Science*, pages 235–239. Springer, 2003.

[15] J. M. E. Hyland and C.-H. L. Ong. On full abstraction for PCF: I, II, and III. *Inf. Comput.*, 163(2):285–408, 2000.

[16] J. Laird. A fully abstract game semantics of local exceptions. In *LICS*, pages 105–114, 2001.

[17] P. Malacaria and C. Hankin. Non-deterministic games and program analysis: An application to security. In *LICS*, pages 443–452, 1999.

[18] C.-H. L. Ong. Observational equivalence of 3rd-order Idealized Algol is decidable. In *LICS*, pages 245–256, 2002.

[19] T. W. Reps, S. Horwitz, S. Sagiv, and G. Rosay. Speeding up slicing. In *SIGSOFT FSE*, pages 11–20, 1994.

[20] M. Weiser. Program slicing. *IEEE Trans. Software Eng.*, 10(4):352–357, 1984.