AGENCE NATIONALE DE LA RECHERCHE ANR	PROGRAMME ARPEGE EDITION 2009	VERIDYC DELIVERABLE 1

Acronyme/Acronym	**VERIDYC**
Titre du projet	Verification des programmes C dynamiques
Proposal title	VERIfication of DYnamic C programs
Deliverable I	Sequential Programs with Simple Data Structures

Table des matières

1 Introduction

During the first year of the project, the members of the consortium have developped two methods for the analysis of sequential programs with infinite data domains of the following types:

1. **lists containing numeric values** (integers, reals). The model used for analysis follows the standard imperative programming paradigm, in which lists can share a common tail, or present circularities. One may consider pointer operations such as dynamic cell creation and delete, destructive updates of selector fields, as well as arithmetic operations on the data contained within the lists. This first method, based on abstract interpretation, synthesizes invariants true at each control point of the program.

2. **integers** (scalars) and **pointers** (arrays). A precise analysis for sequential programs with integer and pointer variables has been developped. The method deals equally with programs with arrays and pointer variables, being able to detect array-out-of-bounds and low-level pointer manipulation errors, such as freeing a non-allocated pointer, or reading from a non-aligned memory location.

A common point of the two techniques above is that they can handle programs with function calls in a compositional way: we compute function summaries and use this information at the calling sites. This method avoids function inlining, which would blow up the size of the models, making the analysis unrealistic. Exploiting the procedural structure of the program is thus of the keys to scalable program verification.

Currently both analyses are implemented in two prototype tools, CINV and FLATA. We have used the FRAMA-C platform for integration, as both tools take as input C code, via two different plugins, CELIA and FLATA-C, respectively. In both cases, the input C program is sliced, using the FRAMA-C slicing plugin, in order to preserve the variables of the data types handled by our analyses. In some cases, using only slicing may not suffice to generate models of manageable size, hence we also use abstraction.

2 Model Extraction: Slicing for sequential C programs

Slicing for sequential C programs is achieved through the use of a Frama-C plug-in. The slicing plug-in relies on the results obtained by two other plug-ins of the platform:

- the value analysis plug-in, that offers (in particular) aliases resolution of the underlying program ;
- the program dependence graph (PDG for short) plug-in.

A slice of program at a given point `p` and a given variable `x` corresponds to the set of statements that might affect the value of said variable `x` at point `p`. This set of statements should still be a compilable and executable program on its own.

The essential part of the slicing plug-in is the ability to construct a sound program dependency graph of the input code. For this reason, let us give here a brief summary of the PDG plug-in.

Program dependence graph computation The PDG plug-in computes different kinds of dependencies related to the analyzed program.

- *value dependency*: in an assignment `x=a+b`, the variable `x` depends on both the values of `a` and `b` before the assignment;
- *address dependency*: in an assignment `*p=x`, there is a dependency between the locations pointed to by `p` and `x`;
- *control dependency*: when a datum can be modified through various execution paths, it depends from the conditions determining the path choice;
- *data dependency*: variables used in an instruction trigger a dependency on their declarations (they indeed must have been declared to be used).

The PDG plug-in computes these dependencies by a forward dataflow analysis on the control-flow graph of the program. Once a PDG for the given program has been computed, the problem of slicing is mostly reduced to the problem of the reachability of a given node in this graph.

Slicing computation The slicing plug-in finely drives the PDG plug-in in order to exploit its analyzes locally. Whereas the PDG plug-in does a general computation, the slicer knows its goals and uses that to adapt the PDG to its needs. Actually, the use of the PDG computation is modular in the context of the slicer.

Also, an effort has been made to keep the analyzes as precise as possible, for example in the context of inter-procedural analyzes.

Slicing criteria Various slicing criteria can be given either regarding code observation or regarding logical properties of the program.

In the case of code observation, several elements can be marked as elements to be preserved in the resulting slice. Slicing can be made on function calls and returns, read and write accesses to selected left-values of the code, or left-values at the end of the entry-point function of the code. Slicing on function returns (e.g., for functions $f_1, ...f_n$) ensures for example that each time these functions do indeed return in the original code, their sliced counterparts also terminate with the same return value. (Frama-C's slicing plug-in is sound in presence of potentially infinite loops, but only with respect to the semantics above: for functions that never terminate, slicing may require special care by the user to keep the reason for non-termination.)

Frama-C's slicing plug-in also has the ability to be used on logical properties specified in ACSL (ANSI/ISO C Specification Language). In this case, properties verified by the

```
/*@ assigns *p \from \empty;
    assigns \result ; */
int scanf (char const *, int * p);
int printf (char const *, int);

int send1 (int x) {
  printf ("%d\n", x) ;
  return x;
}
int send2 (int x) {
  printf ("%d\n", x) ;
  return x;
}
int send3 (int x) {
  printf ("%d\n", x) ;
  return x;
}
int send4 (int x) {
  printf ("%d\n", x) ;
  return x;
}

int fetch(void) {
    static int nb_fetch = 0;
    int value;
    nb_fetch++;
    scanf ("%d",&value);
    return value;
}
```

```
int main(void) {
    int red, green, blue, yellow;
    int sweet,sour,salty,bitter;
    int i;

    red = fetch();
    blue = fetch();
    green = fetch();
    yellow = fetch();

    red = 2*red;
    sweet = red*green;
    sour = 0;
    for (i = 0; i < red; i++)
        sour += green;
    salty = blue + yellow;
    green = green + 1;
    bitter = yellow + green;

    send1 (sweet);
    send2 (sour);
    send3 (salty);
    send4 (bitter);
    return 1;
}
```

Figure 1: Source code of the example

sliced code are ensured to be verified by the initial code. This can be used to slice function assertions, loop variants and invariants or threats emitted by the value analysis plug-in.

The slicing plug-in associates a mark to every instruction of the initial code, which can be also visualized through Frama-C's GUI. The value **bottom** is given to instructions that need not be kept (according to a given slicing criterion). Other instructions, that have been kept but are not needed for the given slicing criterion are marked as **spare**: they are needed for the program to remain compilable. Finally, semantically relevant instructions are marked depending on their dependency types (address, control or data).

Use case example A simple example is given in Figure 1. It is used to demonstrate the slicer's ability to deal correctly with inter-procedural analysis, as well as ACSL annotations.

Assume we want to slice on the return value of function `send1`, or on the calls to function `send2`. Frama-C's slicing plug-in will be invoked respectively by:

```
frama-c  foo.c -slice-print -slice-return send1
frama-c  foo.c -slice-print -slice-calls send2
```

The result of each invocation is shown in Figure 2. In both cases, the calls to `printf` have been deemed irrelevant and removed, as functions without ACSL prototypes are assumed not to change the memory. On the other hand, the specification of the `scanf` function implies that it assigns its second argument, and the relevant calls have been kept. Notice that in the slicing for `send1`, the `for` loop inside the main function is entirely removed, as it is not used to compute `sweet`. Moreover, when slicing on the calls to `send2`, the slicing plugin detects that the argument of `send2` is not useful, and

```
extern int scanf(char const *, int *p ) ;
int send1_slice_1(int x ) {
  return (x);
}

int fetch_slice_1(void) {
  int value ;
  scanf("%d",& value);
  return (value);
}

void main(void) {
  int red ;
  int green ;
  int sweet ;
  red = fetch_slice_1();
  green = fetch_slice_1();
  red = 2 * red;
  sweet = red * green;
  send1_slice_1(sweet);
  return;
}
```

```
extern int scanf(char const *, int *p ) ;
int send2_slice_1(void) {
  return;
}

int fetch_slice_1(void) ;
int fetch_slice_1(void) {
  int value ;
  scanf("%d",& value);
  return (value);
}

void main(void)
{
  int red ;
  int green ;
  int sour ;
  int i ;
  red = fetch_slice_1();
  green = fetch_slice_1();
  red = 2 * red;
  sour = 0;
  i = 0;
  while (i < red) { sour += green; i ++; }
  send2_slice_1(sour);
  return;
}
```

Figure 2: Source code of the example

simplifies the prototype of the function by removing it.

Slicing techniques, extended to a whole program, allows to produce a model of the code, where only the interesting functions have been kept. Currently, the slicing plug-in works only for sequential programs. For concurrent systems programs, we foresee to apply slicing to obtain a model focusing, for example, on lock takes and releases to compute a deadlock analysis (i.e., are we able to reach a deadlocked state from the start of this program ?).

3 Sequential Programs with Lists and Integer Data

Automatic synthesis of valid assertions about programs, such as loop invariants or procedure summaries, is an important and highly challenging problem. The LIAFA team has addressed this problem for sequential programs manipulating *singly-linked lists with unbounded data such as integers or reals.* These programs may contain procedure calls, and actually they are in many cases naturally written using recursive procedures. Examples of such programs are sorting algorithms and programs manipulating sorted lists.

Assertions about these programs typically involve constraints on the shape of the structures, their sizes, the data values contained in the memory cells, the multisets of their data, etc. Consider for instance the well known algorithm `quicksort` that sorts the list pointed by `a` given as input. The specification of `quicksort` includes (1) the sortedness of the output list pointed to by `res`, expressed by the formula:

$$\forall y_1, y_2.\ 0 \leq y_1 \leq y_2 < \mathtt{len}(\mathtt{res}) \Rightarrow \mathtt{data}(y_1) \leq \mathtt{data}(y_2) \qquad (1)$$

where y_1 and y_2 are interpreted as integers and used to refer to positions in the list pointed to by `res`, `len`(`res`) denotes the length of this list, and `data`(y_1) denotes the integer stored in the element of `res` at position y_1, and (2) the preservation property saying that the multiset of data of the input list `a` is equal to the multiset of data of the output list `res`. This property is expressed by

$$\mathtt{ms}(\mathtt{a}^0) = \mathtt{ms}(\mathtt{res}) \tag{2}$$

where $\mathtt{ms}(\mathtt{a}^0)$ (resp. $\mathtt{ms}(\mathtt{res})$) denotes the multiset of integers stored in the list pointed to by `a` at the beginning of the procedure (resp. `res` at the end of the procedure).

We have defined and implemented an approach for automatic assertion synthesis of such constraints based on inter-procedural analysis within the framework of abstract interpretation [12]. More precisely, we consider abstract domains for expressing constraints on relations between program configurations, and we define compositional techniques for computing procedure summaries concerning various aspects such as shapes, sizes, and data.

This work is built on our previous work [3] where we have defined an accurate *intra*-procedural abstract analysis for synthesizing invariants of programs with lists without procedure calls. In this approach, abstract domains are defined where elements are pairs composed of a heap backbone and an abstract data constraint. While the techniques in [3] are strong enough to generate complex invariants for iterative programs, they cannot be applied for compositional computation of procedure summaries.

The extension to the inter-procedural analysis is not trivial due to many delicate problems that appear when addressing the compositionality issue. Indeed, in the spirit of [25], at each procedure call, the callee has only access to the part of the heap that is reachable from its actual parameters. The use of such local heaps is delicate due to the fact that there are relations between the elements of the local heap of the callee, and of the heaps of the procedures that are in the call stack. If these relations are lost during the analysis, this one can be unsound in some cases, or very imprecise in others. However, it is not feasible to maintain explicitly these relations during the analysis.

Thus, a compositional and accurate inter-procedural analysis requires to define an operation for composing abstract domains (e.g., first-order formulas with multiset constraints, or first-order formulas of different types). We have proposed such an operation based on unfolding/folding of lists. This operation can be used, at procedure calls and returns, to (1) compute an over-approximation of the intersection between a first-order formula and a multiset constraint and (2) to convert universal formulas of some kind (e.g., sorting) to formulas of some other kind (e.g. list equality).

Beyond compositional summary computation, the operation we have defined for combining abstract domains allows to tackle two other interesting problems. First, it allows to define a slightweight sound (but not complete) decision procedure for such kinds of formulas, which is useful for carrying out pre-post condition reasoning. Furthermore, our techniques are accurate enough to be used for automatic procedure equivalence checking. It is easy to see that this problem can be reduced to inter-procedural analysis, provided that it is possible to express equality between structures, and derive such properties.

We have implemented our inter-procedural analysis in a plugin called CELIA of the FRAMA-C platform [10] for C program analysis. CELIA checks first that the program is in the class dealt by the analysis (i.e., programs with singly linked lists). Then, it applies the inter-procedural analysis on the inter-procedural control flow graph built by FRAMA-C from the C program. The implementation of CELIA invokes/adapts (1) the heap abstract domains of universal formulas and multi-set constraints provided by the CINV tool [3], (2) the numerical domains of the APRON platform [22], and (3) the generic module of fixpoint computation over control-flow graphs due to B. Jeannet [21]. It has been carried out by implementing in C the abstract transformers including the abstract domain combination/strengthening.

4 Sequential Programs with Integer and Pointer Variables

The current work of the VERIMAG team addresses the problem of verification of certain intrinsic (non-specified) errors in C programs, which involve manipulation of infinite data structures such as integers, arrays and pointers, without recursive data structures (such as lists, trees, etc.). Such programs are commonplace in industrial control software, such as the case study supplied by the EDF team during the first year of the project. Our verification techniques are based on the model of *counter automata*.

Counter automata (equivalently, non-deterministic integer programs) are important models of computation, that can naturally encode many classes of systems with unbounded (or very large) data domains, such as hardware circuits with cache memory, programs with integer variables, integer arrays and recursive data structures. Moreover, recent research has revealed several methods reducing verification problems of several classes of systems to decision problems on counter automata in a more practical way, hence the growing interest for analysis tools working on counter automata. Examples of such systems that can be effectively verified by means of counter automata include: specifications of hardware components [28], programs with singly-linked lists [4, 16, 9], trees [19], and integer arrays [7].

The work of the VERIMAG team aims at reducing the verification of C code to the *reachability problem* of a counter automaton. Currently we are developping a FRAMA-C plugin (called FLATA-C) that translates C programs into counter automata by tracking down the following types of data manipulations:

1. *test and updates of integer variables* are mainly considered in order to obtain a precise representation of the program. Abstracting away integer variables that occur within `if-then-else` and `while` conditional statements would increase the loss of information due to abstraction.

2. *updates of array indices* are represented in our models, in order to detect array-out-of-bounds errors in C programs.

3. *pointer allocation, deallocation and arithmetic* are modeled as operations on integer counters. The extracted models allow to reduce intrinsic program errors such as:

- freeing a pointer variable not referencing an allocated zone,
- accessing data via a non-aligned pointer

to the reachability of an error state in a counter automaton.

The FRAMA-C slicing plugin is used first to strip the source code of irrelevant data types and statements. Slicing is however a very conservative technique which will keep certain variables upon which relevant data depends. In addition to slicing, we use abstraction (i.e., we introduce non-deterministic choices) in order to eliminate completely references to variables of types other than numeric or pointer. The translation to counter automata maintains the functional structure of the program (i.e. each C function is compiled to a different counter automaton).

Given a counter automaton, we aim at computing the relation between the input and the output counter valuations, whenever this relation can be expressed in Presburger arithmetic. In cases where the precise relation cannot be expressed in Presburger arithmetic, we stop the computation and return an under-approximation of the relation. The main application of deriving the input-output relation of a counter automaton in a decidable logic is an algorithm for verifying safety properties of systems, which can be encoded as reachability conditions.

At the heart of our method lies a technique for computing transitive closures of loops labeled by conjunctive transition relations. In general, the transitive closure of linear relations falls outside of known decidable fragments of arithmetic. To this end, it is important to know for which classes of transition relations it is possible to compute the transitive closure precisely and fast – the relations falling outside these classes being dealt with using suitable abstractions. The three main classes of integer relations for which transitive closures can be computed precisely in finite time are: (1) *difference bounds constraints* [11, 6], (2) *octagons* [24, 5], and (3) *finite monoid affine transformations* [2, 15]. For these three classes, the transitive closures can be effectivelly defined in Presburger arithmetic.

We have studied the three non-trivial classes of relations mentioned in the previous and we have shown that they are ultimately periodic, i.e. that each relation R in these classes can be mapped into an integer matrix M_R such that the sequence $\{M_{R^k}\}_{k=0}^{\infty}$ is periodic. The proof of the fact that a sequence of matrices is ultimately periodic relies on a result from tropical semiring theory [27]. This provides shorter proofs to the fact that the transitive closures for these classes can be effectivelly computed, and that they are Presburger definable.

Our algorithm computes the transitive closure of difference bounds and octagonal relations up to four orders of magnitude faster than our original methods from [6, 5], and also scales much better in the number of variables.

Last, we gave a semi-algorithmic method for computing the input-output relation of a counter automaton. The algorithm builds the relation incrementally, by eliminating control states and composing incoming with outgoing relations. The main difficulty here is the elimination of states with several self-loops. We tackle this issue by first computing the transitive closures of the self-loops individually, and then exploring all interleavings between them, until no new relations are discovered. Obviously, this exploration might not end, in which case we stop it at a certain depth and return an underapproximation of the transitive closure.

The techniques reported here are implemented in the FLATA toolset [17] for the analysis of counter automata models. So far, this toolset has been succesfully applied on models that where automatically generated from VHDL specifications of hardware, programs with lists, and verification conditions of programs with integer arrays. We are currently developing a FRAMA-C plugin that will extend the input of the FLATA tool to real-life C programs.

5 Implementation and Test Cases

This section reports on the status of the prototype tools developped by the consortium. Currently there are two analysis tools targetting two classes of sequential C programs:

- the CINV tool, developped by the LIAFA team, for the synthesis of invariants for programs with lists and numeric data
- the FLATA tool, developped by the VERIMAG team, for the verification of programs with integers, arrays and pointers.

Both tools connect to the C language via two Frama-C plugins, namely CELIA for CINV, and FLATA-C for FLATA.

Tool distribution All tools are available for download under LGPL license (`http://www.gnu.org/licenses/lgpl.html`) from the following locations:

- CINV: `http://www.liafa.jussieu.fr/cinv/index.html`
- FLATA: `http://www-verimag.imag.fr/FLATA.html`

5.1 Programs with lists

CELIA has been applied to a benchmark of C programs. Table 1 describes some of our experimental results on the synthesis of procedure summaries. In this table, programs are classified in six classes. The class **sll** includes C functions performing elementary operations on list: *add*ing/*del*eting the *f*irst/*l*ast element, *init*ializing a list of some length. These operations are common in libraries for linked lists, e.g., `linux/list.h` of the Linux distribution. The classes **map** and **map2** include C functions performing a traversal of one resp. two lists, without modifying their structures, but modifying their data. The classes **fold** and **fold2** include C functions computing from one resp. two input lists some output parameters of type list or integer. Finally, the **sort** class includes sorting algorithms on lists. The procedures in classes **map*** and **fold*** are tail recursive, thus we consider for them both iterative and recursive versions. The third column of

class	*fun*	*nesting* (loop,rec)	$\mathbb{M}$ t (s)	$\mathbb{U}$ t (s)	*Examples of summaries synthesized*
sll	*init*	$(0,-)$	<1	<1	
	addfst	–	<1	<1	
	addlst	$(0,1)$	<1	<1	$\rho^{\#}_{\mathbb{U}}(init(\&x,\ell)) : \mathtt{hd}(x)=0 \wedge \mathtt{len}(x)=\ell \wedge \forall y \in \mathtt{tl}(x) \Rightarrow x[y]=0$
	delfst	–	<1	<1	
	dellst	$(0,1)$	<1	<1	
map	*init*(v)	$(0,1)$	<1	<1	$\rho^{\#}_{\mathbb{U}}(init(v,x)) : \mathtt{len}(x^0)=\mathtt{len}(x) \wedge \mathtt{hd}(x)=v \wedge \forall y \in \mathtt{tl}(x).\ x[y]=v$
	initSeq	$(0,1)$	<1	<1	$\rho^{\#}_{\mathbb{U}}(add(v,x)) : \mathtt{len}(x^0)=\mathtt{len}(x) \wedge \mathtt{hd}(x)=\mathtt{hd}(x^0)+v \wedge$
	add(v)	$(0,1)$	<1	<1	$\forall y_1 \in \mathtt{tl}(x), y_2 \in \mathtt{tl}(x^0).\ y_1=y_2 \Rightarrow x[y_1]=x^0[y_2]+v$
map2	*add*(v)	$(0,1)$	<1	<1	$\rho^{\#}_{\mathbb{U}}(add(v,x,z)) : \mathtt{len}(x^0)=\mathtt{len}(x) \wedge \mathtt{len}(z^0)=\mathtt{len}(z) \wedge eq(x,x^0) \wedge$
	copy	$(0,1)$	<1	<1	$\forall y_1 \in \mathtt{tl}(x), y_2 \in \mathtt{tl}(z).\ y_1=y_2 \Rightarrow x[y_1]+v=z[y_2]$
fold	*del*Pred	$(0,1)$	<1	<1	$\rho^{\#}_{\mathbb{M}}(split(v,x,\&l,\&u)) : \mathtt{ms}(x)=\mathtt{ms}(x^0)=\mathtt{ms}(l)\cup\mathtt{ms}(u)$
	max	$(0,1)$	<1	<1	$\rho^{\#}_{\mathbb{U}}(split(v,x,\&l,\&u)) : equal(x,x^0) \wedge \mathtt{len}(x)=\mathtt{len}(l)+\mathtt{len}(u) \wedge$
	clone	$(0,1)$	<1	<1	$l[0] \le v \wedge \forall y \in \mathtt{tl}(l) \Rightarrow l[y] \le v \wedge$
	split	$(0,1)$	<1	<1	$u[0] > v \wedge \forall y \in \mathtt{tl}(u) \Rightarrow u[y] > v$
fold2	*equal*	$(0,1)$	<1	<1	$\rho^{\#}_{\mathbb{M}}(merge(x,z,\&r)) : \mathtt{ms}(x)\cup\mathtt{ms}(z)=\mathtt{ms}(r) \wedge \mathtt{ms}(x^0)=\mathtt{ms}(x) \wedge \ldots$
	concat	$(0,1)$	<1	<3	$\rho^{\#}_{\mathbb{U}}(merge(x,z,\&r)) : sorted(x^0) \wedge sorted(z^0) \wedge sorted(r) \wedge$
	merge	$(0,1)$	<1	<3	$equal(x,x^0) \wedge equal(z,z^0) \wedge \mathtt{len}(x)+\mathtt{len}(z)=\mathtt{len}(r)$
sort	*bubble*	$(1,-)$	<1	<3	
	insert	$(1,-)$	<1	<3	$\rho^{\#}_{\mathbb{M}}(quicksort(x)) : \mathtt{ms}(x)=\mathtt{ms}(x^0)=\mathtt{ms}(res)$
	quick	$(-,2)$	<2	<4	$\rho^{\#}_{\mathbb{U}}(quicksort(x)) : equal(x,x^0) \wedge sorted(res)$
	merge	$(-,2)$	<2	<4	

Table 1: Experimental results for functions in the benchmark for lists.

Table 1 specifies the versions considered (iterative/recursive) and the number of nested loops or recursive calls.

Columns 4–5 provide (upper bounds on) the running time of the analysis for the multiset abstract domain (denoted by $\mathbb{M}$) and the universal formulas domain (denoted by $\mathbb{U}$). Column 6 shows samples of procedure summaries (denoted by $\rho^{\#}$) that CELIA can synthesize with each abstract domain. All experiments have been done on an Intel686 with 4GHz and 4Go of RAM.

5.2 Counter Automata

This section reports on experiments we have performed with the FLATA toolset, to decide reachability in several counter automata, modeling real-life systems. We have experimented on three sets of counter automata. The first set of models we considered is taken from [28], where an approach for verification of generic VHDL circuit designs based on translation to counter automata is presented. Traditional verification techniques for hardware systems usually assume that the state space of these systems is finite. The approach presented in [28] aims at verification of parameterized VHDL components with infinite state space. These generic components allow for the creation of libraries of reusable hardware, and are therefore commonplace in practice.

The translation to counter automata described in [28] maps bit variables to control locations, and integer variables to counters. Various safety properties are encoded as bit variables whose values are equivalent to propositional logic formulae representing the bad (unsafe) states.

We report on three out of the total five models from [28] (the remaining two are currently beyond the capabilities of FLATA and are a source of motivation for our future work). For the component of a hardware `counter`, we check whether overflow of a parametric bound is possible. For the `register` component, we check if the reset works correctly. The `synlifo` is a synchronous LIFO component with push and pop operations, which implements signals empty and full. The property checks if these signals are set correctly for a LIFO container of arbitrary size. Additionally, we have created buggy versions of these models (`counter-bug`, `register-bug`, and `synlifo-bug`).

The second set of examples are counter automata generated from programs with singly-linked lists, using the approach described in [8]. The main idea is that the set of heaps generated by a program with a finite number of local variables can be represented by a finite number of shape graphs, and the (unbounded) lengths of various list segments can be tracked by counters. The result of the translation of a program with lists is a counter automaton whose transition semantics is in bisimulation with the original program.

For all singly-linked list programs, we check that there are no null pointer dereferences. The `InsDel` program inserts elements to an empty list non-deterministically many times while counting the number of insertions and then traverses this number of elements and deletes each traversed element. Here we also check whether the list is empty when the program ends. The `ListCounter` program counts the length of a list during a first traversal of the list, which increments a counter. Here we also check that the length of the list is unchanged, and equal to the final value of the counter. The `ListReversal` is a textbook program that returns a list containing the same elements as the input list, in the reversed order. The reversal is done in place, by changing the links between the cells, instead of creating a copy of the input list. Here we also check that the lengths of the input list equals the length of the output list.

The third set of counter automata models are given by the decision procedure of the array logic SIL (Singly Indexed Logic), described in [20]. The decidability of the satisfiability problem for SIL encodes the set of models of a formula as the union of sets of traces of a set of flat counter automata with difference bounds constraints, whose emptiness is known to be decidable e.g., [11, 15]. Since FLATA is guaranteed to terminate on flat models with ultimately periodic relations on loops, we can use it as a solver for the SIL logic.

We report on two SIL formulae which arise as verification conditions for loop invariants of array manipulating programs. The *array rotation* program rotates an array by one element to the left and the *array split* program splits an array to negative and non-negative parts. Both formulae are translated to a pair of counter automata, hence Table 2 reports the results of analyses as sum of the number of states, transition and verification times for both automata.

We have compared the performance of FLATA with other existing tools for the analysis of counter automata. The FAST tool [1] is based on the acceleration of loops with finite monoid affine transformations – we have used FAST with four different plugins for solving

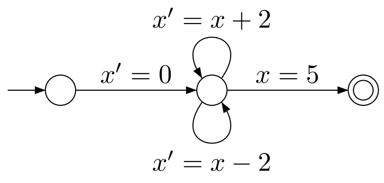

Figure 3: The modulo counter automaton.

model	$\|Q\|$	$\|T\|$	FLATA	ARMC	FAST		ASPIC
					MONA	PRESTAF	
counter	6	13	0.8	0.1	0.7	0.7	0.1
counter-bug	6	13	0.9	0.1	0.7	0.7	D
register	10	49	0.9	0.1	0.14	0.5	0.1
register-bug	10	49	0.9	0.1	0.14	0.5	D
synlifo	43	1006	25	1.5	500	112	1.7
synlifo-bug	43	1006	15	1.5	419	101	D
insdel-bug	31	37	0.4	0.2	225.3	21.4	D
listreversal	97	107	6.2	42.6	T	E	0.2
listreversal-bug	99	107	2.4	0.2	T	T	D
listcounter	31	35	1.0	2.9	178.7	14	0.1
listcounter-bug	31	34	0.5	0.1	T	T	D
split	61	329	9.6	1.2	C	C	C
rotation	33	147	5.7	0.6	C	C	C
modulo	3	4	0.7	T	0.1	0.4	D

Table 2: Reachability comparison (time in seconds). T – time-out 14min, E – segmentation fault, D – "don't know", C – incompatible (non-deterministic) relations

Presburger queries: MONA [23] (finite automata), Prestaf [13] (shared automata), Omega [14] (quantifier elimination) and LASH [29] (numeric decision diagrams). The ARMC tool [26] uses predicate abstraction and interpolation-based abstraction refinement. The ASPIC tool [18] uses widening-based abstract interpretation. Table 2 gives the performance comparison of various tools for the reachability analysis of counter automata on the set of examples described above.

One can notice from the times reported in Table 2 that tools based on abstraction (ARMC, ASPIC) are in general faster than the ones based on precise acceleration of loops (FAST, FLATA), the reason being that abstraction greatly simplifies the models used in the analysis. On the other hand, the imprecision introduced often leads to "don't know" answers (ASPIC) or extra iterations of the abstraction-refinement loop (ARMC).

Another problem of the methods purely based on abstraction is the `modulo` counter automaton depicted in Figure 3. In order to prove that this automaton is empty, one must prove that "x is even" is an invariant of the system. In our experience, FAST and FLATA could prove this property, whereas ARMC and ASPIC could not (cf. Table 2).

6 Ongoing and Future Work

As a short-term goal, we plan on finishing the implementation and testing of the tools on larger C programs, including examples provided by the EDF partner, as well as from the public domain, e.g. the LINUX distribution. We also plan a tighter integration of the tools, by using FLATA as a back end for the CINV tool, in order to explore the use of acceleration techniques to extract more precise invariants for programs with lists.

In particular, the VERIMAG team plans to investigate the extension of compositional verification techniques from sequential to concurrent counter models. These results will be applied to multithreaded C programs with integer and pointer (array) variables.

On the LIAFA side, further work includes (A) the generalization of their framework to structures such as multi-linked lists, trees, and nested structures, and (B) the use of program slicing in order to accept more programs than the ones using only dynamic data structures.

The CEA has been working on a Frama-C plugin for the study of concurrent code. The plug-in uses a least-fixpoint approach to combine sequential analyses of the various threads, in order to obtain a safe approximation of the concurrent behaviour of the program. Once this is done, all statements relevant for the concurrent behaviour will be known, and it will be possible to slice on those statements.

References

[1] S. Bardin, J. Leroux, and G. Point. Fast extended release. In *CAV'06*, volume 4144, pages 63–66. Springer Verlag, 2006.

[2] B. Boigelot. *Symbolic Methods for Exploring Infinite State Spaces*, volume PhD Thesis, Vol. 189. Collection des Publications de l'Université de Liège, 1999.

[3] A. Bouajjani, C. Drăgoi, C. Enea, A. Rezine, and M. Sighireanu. Invariant synthesis for programs manipulating lists with unbounded data. In *CAV*, LNCS, pages 72–88, 2010.

[4] Ahmed Bouajjani, Marius Bozga, Peter Habermehl, Radu Iosif, Pierre Moro, and Tomás Vojnar. Programs with lists are counter automata. In *CAV*, pages 517–531, 2006.

[5] M. Bozga, C. Gîrlea, and R. Iosif. Iterating octagons. In *TACAS '09*, pages 337–351. Springer, 2009.

[6] M. Bozga, R. Iosif, and Y. Lakhnech. Flat parametric counter automata. *Fundamenta Informaticae*, 91:275–303, 2009.

[7] Marius Bozga, Peter Habermehl, Radu Iosif, Filip Konecný, and Tomás Vojnar. Automatic verification of integer array programs. In *CAV*, pages 157–172, 2009.

[8] Marius Bozga, Radu Iosif, and Swann Perarnau. L2ca. `http://www-verimag.imag.fr/L2CA.html`.

[9] Marius Bozga, Radu Iosif, and Swann Perarnau. Quantitative separation logic and programs with lists. In *IJCAR*, pages 34–49, 2008.

[10] CEA. *Frama-C Platform*. `http://frama-c.com`.

[11] H. Comon and Y. Jurski. Multiple Counters Automata, Safety Analysis and Presburger Arithmetic. In *CAV '98*, volume 1427 of *LNCS*, pages 268 – 279. Springer, 1998.

[12] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.

[13] J.M. Couvreur. Prestaf. `http://altarica.labri.fr/forge/projects/3/wiki/PresTAF`.

[14] William Pugh et. al. Omega. `http://www.cs.umd.edu/projects/omega/`.

[15] A. Finkel and J. Leroux. How to compose presburger-accelerations: Applications to broadcast protocols. In *FST TCS '02*, pages 145–156. Springer, 2002.

[16] Alain Finkel, Étienne Lozes, and Arnaud Sangnier. Towards model-checking programs with lists. In *ILC*, pages 56–86, 2007.

[17] http://www-verinew.imag.fr/FLATA.html.

[18] Laure Gonnord. Aspic. `http://laure.gonnord.org/pro/aspic/aspic.html`.

[19] Peter Habermehl, Radu Iosif, Adam Rogalewicz, and Tomás Vojnar. Proving termination of tree manipulating programs. In *ATVA*, pages 145–161, 2007.

[20] Peter Habermehl, Radu Iosif, and Tomáš Vojnar. A logic of singly indexed arrays. In *Proceedings of the 15th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR '08, pages 558–573, 2008.

[21] B. Jeannet. *Fixpoint*. `http://gforge.inria.fr/`.

[22] B. Jeannet and A. Miné. Apron: A library of numerical abstract domains for static analysis. In *CAV*, volume 5643 of *LNCS*, pages 661–667. Springer, 2009.

[23] Nils Klarlund and Anders Møller. Mona. `http://www.brics.dk/mona/`.

[24] A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

[25] N. Rinetzky, J. Bauer, T.W. Reps, S. Sagiv, and R. Wilhelm. A semantics for procedure local heaps and its abstractions. In *POPL*, pages 296–309, 2005.

[26] Andrey Rybalchenko. Armc. `http://www7.in.tum.de/~rybal/armc/`.

[27] B. De Schutter. On the ultimate behavior of the sequence of consecutive powers of a matrix in the max-plus algebra. *Linear Algebra and its Applications*, 307:103–117, 2000.

[28] Ales Smrcka and Tomás Vojnar. Verifying parametrised hardware designs via counter automata. In *Haifa Verification Conference*, pages 51–68, 2007.

[29] P. Wolper and B. Boigelot. Verifying systems with infinite but regular state spaces. In *CAV '98*, pages 88–97. Springer, 1998.