	PROGRAMME ARPEGE EDITION 2009	VERIDYC DELIVERABLE 2
---	----------------------------------	--------------------------

Acronyme/Acronym	<b>VERIDYC</b>
Titre du projet	Verification des programmes C dynamiques
Proposal title	VERification of DYnamic C programs
Deliverable II	Concurrent Programs with Simple Data Structures / Sequential Programs with Composite Data Structures

## Table des matières

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Static Analysis of Concurrent Programs</b>	<b>2</b>
2.1	General architecture . . . . .	3
2.2	Precision . . . . .	4
2.2.1	Context-sensitivity . . . . .	4
2.2.2	“Happened-after” relation . . . . .	4
2.3	Exploiting the locked mutexes information . . . . .	4
2.4	Slicing of concurrent programs . . . . .	5
<b>3</b>	<b>Models of Concurrency</b>	<b>6</b>
3.1	Concurrent Programs with Singly-Linked Lists . . . . .	6
3.1.1	Programs that can be analyzed . . . . .	6
3.1.2	Example . . . . .	7
3.1.3	How Cheap can be used . . . . .	8
3.1.4	Distribution . . . . .	8
3.1.5	Internals . . . . .	8
3.2	Analysis of Recursively Parallel Programs . . . . .	9
<b>4</b>	<b>Sequential Programs with Complex Data Structures</b>	<b>11</b>
4.1	Program Verification using Forest Automata . . . . .	11
4.2	Doubly-linked Lists with Integer Values . . . . .	13
4.3	Shallow Pointers and Pointer Arithmetic . . . . .	15
4.3.1	Example . . . . .	16
4.3.2	Distribution and Usage of FLATA-C . . . . .	17

<b>5 Industrial Test Cases</b>	<b>17</b>
5.1 Rotating Buffer . . . . .	17
5.2 MTHREAD Test Case . . . . .	17
<b>6 Ongoing and Future Work</b>	<b>19</b>

## 1 Introduction

During the second year of the project, the members of the consortium have developed several methods for the verification of the following types of programs:

1. *Concurrent programs with simple data structures*: work on concurrent programs has been pursued along three complementary directions, detailed in the following.
  - (a) Efficient model extraction techniques based on static analysis, including a general library-independent value analysis for multithreaded programs (the MTHREAD plugin)
  - (b) Proof checking of concurrent programs handling singly-linked lists; besides memory consistency, this work targets the detection of race conditions of un-joined threads (the CHEAP plugin)
  - (c) A theoretical study of the complexity of some verification problems within alternative concurrent models (e.g. CILK, X10, etc.)
2. *Sequential programs with composite data structures*: work on sequential programs with complex data structures has been pursued along three complementary directions.
  - (a) An efficient symbolic representation based on *forest automata* and tool support for C programs with tree-like data structures (e.g. tree with head pointers, etc.)
  - (b) Extension of the CELIA plugin to doubly-linked lists with integer data values
  - (c) Extended tool support for programs with *shallow pointers* and *pointer arithmetic* via the common NUMERICAL TRANSITION SYSTEMS interface (the FLATA-C plugin)

Concerning the experimental part, we have considered a new concurrent example in which several (reader and writer) processes communicate using a buffer-like data structure implemented using arrays and pointer arithmetic. This example is currently analysed by the FLATA-C plugin.

## 2 Static Analysis of Concurrent Programs

During the course of the second year, the CEA has developed a FRAMA-C plugin, MTHREAD, to handle concurrent code. This plugin reuses FRAMA-C's existing abstract-interpretation based plugin, VALUE, and extends it to obtain an over-approximation of the behavior of multi-threaded programs.

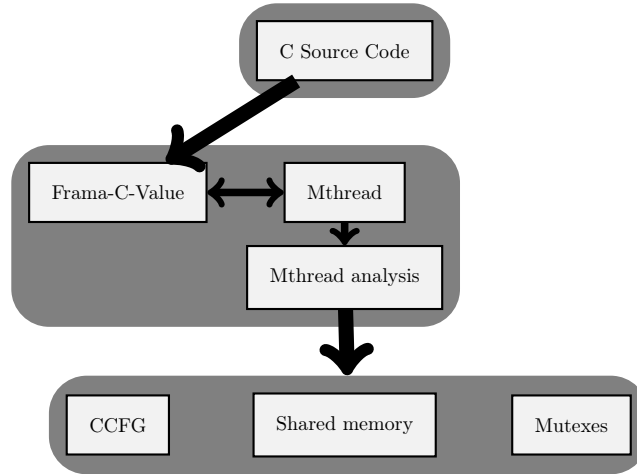


Figure 1: High-level overview of MTHREAD

## 2.1 General architecture

Figure 1 presents a high-level overview of MTHREAD. The C input code is parsed by FRAMA-C, as for any plugin. Then, MTHREAD analyzes the program. Each thread is analyzed independently by Frama-C's Value Analysis.<sup>1</sup> This produces an over-approximation of the sequential behavior of the threads, by construction. However, this is not a correct over-approximation of the concurrent behavior of the program, as the interactions between the threads are not taken into account. Thus, MTHREAD confronts the various sequential approximations, to discover:

1. all the memory locations that are accessed by at least two threads;
2. the messages that are sent by explicit message-passing, and on which message queue those messages are sent;
3. the mutexes that are locked when the above multithreaded events occur.

Using type 1 and 2 information, we re-analyze all the threads that read shared memory, or received messages. This time, we take into account the concurrent data that has been emitted during the first analysis of the threads. Shared zones are marked as volatile, which is a correct (albeit a bit imprecise) approximation. Messages sent on a queue  $q$  by a thread  $t$  are received each time a thread  $t' \neq t$  reads on  $q$ .

If no new data is written or sent during this second analysis, we have obtained an over-approximation of the concurrent behavior of the program. If this is not the case, we repeat the process above until a fixpoint is reached. Convergence is ensured because:

- there are only a finite possible set of shared zones, as the Value Analysis does not handle fully general dynamic allocation;
- if needed, the contents of emitted messages are over-approximated using the widening operation already present in the sequential analysis.

<sup>1</sup> Starting from the main thread. Threads are discovered automatically, each time VALUE analyzes a call to a thread creation primitive.

At the end of the analysis, the plugin emits three outputs:

1. a *concurrent control-graph*, which is similar to a very aggressive slicing of the program; see §2.4 and §5.2 for more details;
2. all the memory zones that are shared amongst multiple threads, with detailed information on which functions access them;
3. the mutex information described in §2.3.

## 2.2 Precision

The approach presented above is always sound, as it produces a correct over-approximation of the behaviors of all the threads of the program. However, it is also important to remain precise. We have thus implemented two optimizations, presented below.

### 2.2.1 Context-sensitivity

It is crucial to have a context-sensitive analysis, meaning that a function  $f$  called by two functions  $g_1$  and  $g_2$  is analyzed separately in the two calling contexts. This is particularly important because embedded code often encapsulates calls to concurrency primitives within its own functions, typically to simplify return codes.

MTHREAD is fully context-sensitive: as soon as a function is reached through two different paths (that is, two differing Value Analysis call-stacks), two different analyzes contexts are created by MTHREAD. This is consistent with what is done by VALUE, which proceeds by recursive inlining.

### 2.2.2 “Happened-after” relation

MTHREAD tries to sequence events, so that if  $e_1$  is guaranteed to always occur before  $e_2$ , then  $e_2$  will not influence the analysis of  $e_1$ . For now, MTHREAD takes into account the fact that a thread has not yet been created, or has been canceled. In the following example, the zone  $v$  is detected as *not* shared, as the possibly concurrent access in  $t_0$  occurs before  $t_1$  has been created.

$$t_0 : \left| \begin{array}{l} v = 1; \\ a = v; \text{ // Only possible value for a: } v \\ \text{create}(t_1); \end{array} \right| t_1 : | v == 2;$$

In the future, we will try to enrich this “happened-after” relation so that it handles condition variables, as those are often used in concurrent C code.

## 2.3 Exploiting the locked mutexes information

The mutexes information collected by MTHREAD is, for now, informative only — that is, it does not influence the other analyses. However, the user can use it to check that a shared memory zone is consistently protected by at least one mutex each time it is accessed in a concurrent setting. A typical result is given below.<sup>2</sup>

<sup>2</sup> This result is extracted from the example presented in §5.2.

```
[bufTSDU.bufHead] write unprotected, read protected by (?)&bufTSDU+24

read by _main_ at sources/Bufertournant.c:399, unprotected,
// Buf_Free (sources/Analyseur.c:496) <-
  Stop_Analyseur (sources/Analyseur.c:744) <-
  main (sources/Analyseur.c:578)

read by &srcThreadTSDU at sources/Bufertournant.c:265, protected by
&bufTSDU+24,
// Buf_Get (sources/tsdu.c:94) <-
  srcThreadTSDU (sources/tsdu.c:72)

write by _main_ at sources/Bufertournant.c:400, unprotected,
// Buf_Free (sources/Analyseur.c:496) <-
  Stop_Analyseur (sources/Analyseur.c:744) <-
  main (sources/Analyseur.c:578)
```

This output means that the field `bufHead` of the global variable `bufTSDU` is accessed concurrently by the main thread and the thread `srcThreadTSDU`. `MTHREAD` indicates that the second thread locks a mutex, named `&bufTSDU+24` here, when it accesses the field. However, this is not the case of the main thread. Hence there might be a race condition when accessing this memory zone. Notice that fully contextual call-stacks are given when describing each access, to ease user comprehension.

The results of our mutex analysis is similar to the one of `LOCKSMITH` [?], and our approach is similar to theirs: identifying shared memory, propagating locked mutexes along the control-flow graph, *etc.* However, the methods differ significantly. `LOCKSMITH` uses dedicated analyzes, that are specialized for this task. This ensures that their analysis is fast. On the contrary, `MTHREAD` reuses most of the machinery of Frama-C's Value Analysis, which usually gives more precise results, as well as a wealth of other information.

## 2.4 Slicing of concurrent programs

Once it has reached a fixpoint, `MTHREAD` automatically builds a *concurrent control-flow graph* (CCFG) for each thread of the program. The CCFG is an inter-procedural control-flow graph, in which each function is inlined at its call point. However, we restrict the CCFG to concurrent events, that are defined as

- a call to a concurrent primitive: thread creation, mutex lock or release, message sending *etc.*
- a read or write access to a shared memory zone.

More precisely, we remove from the CCFG every subgraph that does not contain a concurrent event. In the resulting graph, we also remove non-concurrent nodes that do not modify the control flow. With those restrictions, CCFG are usually quite small, and very readable. An example is given in §5.2.

Using the CCFG it is possible to slice the program to keep only the instructions relevant for the concurrent behavior. For a thread  $t$ , the slicing criterion is simply the following:

*Keep all the source code instructions that gave rise to a concurrent event within the CCFG.*

Calling Frama-C's sequential slicing on the final Value Analysis of  $t$ , with this criterion, results in a program with the following characteristics:

- all function calls to a concurrent primitive reachable by  $t$  are kept — by construction of the criterion;
- all instructions that are useful for computing sent messages, or values inside shared memory, are kept — by definition of a correct slicing.

Thus, we obtain a correct slicing of the program for each thread.

## 3 Models of Concurrency

### 3.1 Concurrent Programs with Singly-Linked Lists

Cheap is a FRAMA-C plugin. It is based on the Heap-Hop tool [?] developed at LSV. It is a proof checker for concurrent heap-manipulating programs written in C and annotated with pre/post conditions, loop invariants, and lock invariants. It guarantees the absence of errors with respect to heap manipulation (including memory leaks), the absence of race conditions, and the absence unjoined threads. It does not provide definite guarantees about progress, but may help discover some sources of deadlocks (see below).

#### 3.1.1 Programs that can be analyzed

Cheap targets the programs that may manipulate heap-allocated recursive data structures (lists, doubly-linked lists, or trees), create and join threads, and synchronize through heap-allocated locks. For simplicity, Cheap assumes a unique type of cell.

```

struct _cell {
    int val;
    int lock; // <- reserved
    struct _cell *tl;
    struct _cell *left;
    struct _cell *right;
};
typedef struct _cell cell;

```

Cell fields can be dereferenced and updated using standard C syntax, at the exception of the `lock` field, which is assumed to be a reserved field (hence the type of this field should be thought as some abstract type `lock` instead of `int`). The only other operations on cells that are currently handled by Cheap are the following ones

```

cell *new();
void dispose(cell *c);
const cell *nil = (cell *)0;

void init_lock(cell *x);
void finalize_lock(cell *x);
void lock(cell *x);
void unlock(cell *x);

int spawn(cell *f(cell *),
          cell *c);
cell *join(int tid);

```

When a cell `c` is allocated by a call `c=new()`, the field `c->lock` does not refer to a valid lock. After a call to `init_lock(c)`, a lock has been allocated in the heap and has been associated to the cell `c` (it is morally pointed to by `c->lock`). Before disposing a cell `c`, its associated lock should be disposed as well by a call to `finalize_lock(c)`. Cheap makes some assumptions on the lock library: locks are not reentrant, and they can be released by a thread even if it was acquired by a different thread. Cheap however prevents releasing non-acquired locks. For instance, `lock(c);lock(c)` always deadlocks, whereas `(lock(c);lock(c))||unlock(c)` may sometimes terminate, but can't be accepted by Cheap whatever the annotations are.

A call to `spawn(f,c)` creates a new thread that runs the function `f` with parameter `c`, and returns the thread identifier (`tid`) of this new thread to the caller. The `tid` can later be used by any other thread to wait for the termination of this thread, obtaining then the return value of `f(c)`. If the return value of `spawn` is ignored by the caller, it is assumed that the spawned thread is detached, and one does not report an error for it not being joined.

### 3.1.2 Example

We have used this cell structure and these primitives to implement a fine-grained concurrent multi-set. The multi-set is represented by an ordered list; it is possible to add a value, to remove it, or to check if it belongs to the multi-set. All of these actions may happen concurrently on disjoint regions of the list: in a first phase, a traversal of the list with hand-over-hand locking determines the region of the list where the action has to occur, then in a second phase the specific action is performed.

```


cell *locate(int i) {
    cell *pred = Head;
    lock(pred);
    cell *cur = pred->t1;
    lock(cur);
    while (cur->val < i) {
        unlock(pred);
        pred = cur;
        cur = cur->t1;
        lock(cur);
    }
    return pred;
}

int membership(int i) {
    cell *pred = locate(i);
    cell *cur = pred->t1;
    int res;
    if (cur->val == i) res = 1;
    else res = 0;
    unlock(pred);
    unlock(cur);
    return res;
}

void add(int e) {
    cell *pred = locate(e);
    cell *cur = pred->t1;
    if (cur->val != e) {
        cell *c = new();
        c->val = e;
        c->t1 = cur;
        pred->t1 = c;
        init_lock(c);
        unlock(c);
    }
    unlock(pred);
    unlock(cur);
}

void remove(int e) {
    cell *pred = locate(e);
    cell *cur = pred->t1;
    if (cur->val == e) {
        pred->t1 = cur->t1;
        finalize_lock(cur);
        dispose(cur);
    } else unlock(cur);
    unlock(pred);
}

```

	PROGRAMME ARPEGE EDITION 2009	VERIDYC DELIVERABLE 2
---	----------------------------------	--------------------------

### 3.1.3 How Cheap can be used

The current version of Cheap is a stand-alone FRAMA-C plugin. It can be called on command-line, specifying which C file should be analyzed. It analyzes each function independently, and returns an error message for each function for which it found that the given annotations were incorrect. The error message explains why the symbolic execution failed (see section “Internals” below), and which line of the original C file corresponds to the step of the symbolic execution that failed. The current version of Cheap generates the following output on the annotated version of the concurrent multi-set described above:

```
./frama-c-CheapPlugin.byte examples/lock-coupling-lists.c
[kernel] preprocessing with
      "gcc -C -E -I. examples/lock-coupling-lists.c"
[cheap] If the specs are Valid (see end of output),
      the program will enjoy the following properties:
      + there is no memory fault or race
      + there is no leak

      Function remove

      Function membership

      Function locate

      Function add
[cheap] Valid
```

### 3.1.4 Distribution

Cheap is under final cleaning. It is planned to be released in June 2012, with a list of case studies and a short tutorial. The exact public license has not been determined yet; the best choice seems the GNU LGPL v2 (like FRAMA-C), but the code is a derivative of Heap-Hop, which, for historical reasons, is under QPL. But it seems that QPL forces then to distribute the modified source code of Heap-Hop as *patches*. Since Cheap source code does not contain FRAMA-C (it is only a plugin), the choice of the QPL would make sense as well, as a branch of Heap-Hop.

### 3.1.5 Internals

Cheap transforms an annotated C program in a set of verification conditions of the form  $\{pre\}p\{post\}$  where  $p$  is a loop-free sequential program. A symbolic execution is then executed for each verification condition – this is mostly an incomplete calculus of strongest post-condition. The execution may fail before the strongest post-condition of the entire  $p$  has been computed, because at some intermediate step the precondition cannot guarantee that the next instruction is safe.



The language of annotations used by Cheap is an adaption of the one of Heap-Hop that respects the syntactic conventions of ACSL. Although it is accepted by the FRAMA-C parser for ACSL, it does not follow the informal semantics of ACSL. A fundamental difference is that the conjunction symbol `&&` is understood by Cheap as the separating conjunction of separation logic. The Cheap language is far less expressive than the ACSL language. It assumes only a small set of basic predicates dealing with heap structures, threads, and locks, and cannot handle any user-defined predicate (in particular, recursive ones).

The symbolic execution of Heap-Hop has been enhanced in several ways: for instance, it now supports derouting C primitives (`break`, `return`, `continue`), and it can deal with heap-allocated locks and threads (whereas the closest features Heap-Hop could handle were conditional critical sections and a simple form of parallel function calls).


A good introduction to the techniques used by Cheap and Heap-Hop can be found in the papers on the Smallfoot tool [?, ?] for the most general aspects, and in a paper by Gotsman & al [?] for the specific treatment of locks and threads. The Verifast tool [?] also offers a support for thread creation and heap-allocated locks: it is based on a richer language of annotations, but it requires a larger amount of annotations in order to guide the symbolic execution.

### 3.2 Analysis of Recursively Parallel Programs

Despite the ever-increasing importance of concurrent software (e.g., for designing reactive applications, or parallelizing computation across multiple processor cores), concurrent programming and concurrent program analysis remain challenging endeavors. The most widely available facility for designing concurrent applications is *multithreading*, where concurrently executing sequential threads nondeterministically interleave their accesses to shared memory. Such nondeterminism leads to rarely-occurring “Heisenbugs” which are notoriously difficult to reproduce and repair. To prevent such bugs programmers are faced with the difficult task of preventing undesirable interleavings, e.g., by employing lock-based synchronization, without preventing benign interleavings—otherwise the desired reactivity or parallelism is forfeited.

The complexity of multi-threaded program analysis seems to comply with the perceived difficulty of multi-threaded programming. The state-reachability problem for multi-threaded programs is PSPACE-complete [?] with a finite number of finite-state threads, and undecidable [?] with recursive threads. Current analysis approaches either explore an underapproximate concurrent semantics by considering relatively few interleavings [?, ?] or explore a coarse overapproximate semantics via abstraction [?, ?].

Explicitly-parallel programming languages have been advocated to avoid the intricate interleavings implicit in program syntax [?], and several such industrial-strength languages have been developed [?, ?, ?, ?, ?, ?]. Such systems introduce various mechanisms for creating (e.g., “fork”, “spawn”, “post”) and consuming (e.g., “join”, “sync”) concurrent computations, and either encourage (through recommended programming practices) or ensure (through static analyses or runtime systems) that parallel computa-

	PROGRAMME ARPEGE EDITION 2009	VERIDYC DELIVERABLE 2
---	----------------------------------	--------------------------

tions execute in isolation without interference from others, through data-partitioning [?], data-replication [?], functional programming [?], message passing [?], or version-based memory access models [?], perhaps falling back on transactional mechanisms [?] when complete isolation is impractical. Although few of these systems behave deterministically, consuming one concurrent computation at a time, many are sensitive to the order in which multiple isolated computations are consumed. Furthermore, some allow computations creating an unbounded number of sub-computations, returning to their superiors an unbounded number of handles to unfinished computations. Even without multithreaded interleaving, nondeterminism in the order in which an unbounded number of computations are consumed has the potential to make program reasoning complex.

In this work, we (**group of LIAFA**) investigate key questions on the analysis of interleaving-free programming models. Specifically, we ask to what extent such models simplify program reasoning, how those models compare with each other, and how to design appropriate analysis algorithms. We attempt to answer these questions as follows:

- We introduce a general interleaving-free parallel programming model on which to express the features found in popular parallel programming languages.
- We discover a surprisingly-complex feature of some existing languages: even simple classes of programs with the ability to pass unfinished computations both to and from subordinate computations have undecidable state-reachability problems.
- We show that the concurrency features present in many real-world programming languages such as Cilk, X10, and Multilisp are captured precisely (modulo the possibility of interleaving) by various fragments of our model.
- For fragments corresponding to real-world language features, we measure the complexity of computing state-reachability for finite-data programs, and provide, in most cases, asymptotically optimal state-reachability algorithms.

Our focus on finite-data programs without interleaving is a means to measuring complexity for the sake of comparison, required since state-reachability for infinite-data or multi-threaded programs is generally undecidable. Applying our algorithms in practice may rely on data abstraction [?], and separately ensuring isolation [?], or approximating possible interleavings [?, ?, ?, ?]; still, our handling of computation-order non-determinism is precise.

The major distinguishing language features are whether a single or an arbitrary number of subordinate computations are waited for at once, and whether the scope of subordinate computations is confined. Generally speaking, reasoning for the “single-wait” case is less difficult than for the “multi-wait” case, and we demonstrate a range of complexities<sup>3</sup> from PTIME, NP, EXPSPACE, and 2EXPTIME for various scoping restrictions. Despite these worst-case complexities, a promising line of work has already demonstrated effective algorithms for practically-occurring EXPSPACE-complete state-reachability problem instances based on simultaneously computing iterative under- and over-approximations, and rapidly converging to a fixed point [?, ?].

---

3. In order to isolate concurrent complexity from the exponential factor in the number of program variables, we consider a fixed number of variables in each procedure frame; this allows us a PTIME point-of-reference for state-reachability in recursive sequential programs [?].

We thus present a classification of concurrency constructs, connecting programming language features to fundamental formal models, which highlight the sources of concurrent complexity resulting from each feature, and provide a platform for comparing the difficulty of formal reasoning in each. We hope that these results may be used both to guide the design of impactful program analyses, as well as to guide the design and choice of languages appropriate for various programming problems.

## 4 Sequential Programs with Complex Data Structures

### 4.1 Program Verification using Forest Automata

During the second year of the project, the LIAFA team (Peter Habermehl) as well as VERIMAG (Jiri Simacek) have worked on a method for verifying sequential programs with complex *dynamic linked data structures* such as various forms of singly- and doubly-linked lists (SLL/DLL), possibly cyclic, shared, hierarchical, and/or having different additional (head, tail, data, and the like) pointers, as well as various forms of trees. We in particular consider C pointer manipulation, but our approach can easily be applied to any other similar language. For example we can handle the Deutsch-Schorr-Waite tree traversal algorithm which traverses a tree by just redirecting pointers without marking nodes.

```

/*
 * The Deutsch-Schorr-Waite tree traversal algorithm
 */
#include <stdlib.h>

int __nondet();

struct TreeNode {
    struct TreeNode* left;
    struct TreeNode* right;
};

struct TreeNode* createTree() {

    struct TreeNode* root = malloc(sizeof(struct TreeNode));
    root->left = NULL;
    root->right = NULL;
    struct TreeNode* curr;
    while (__nondet()) {
        curr = root;
        while (curr->left && curr->right)
            curr = (__nondet())?(curr->left):(curr->right);

        if (!curr->left && __nondet()) {
            curr->left = malloc(sizeof(struct TreeNode));
            curr->left->left = NULL;
            curr->left->right = NULL;
        }
        if (!curr->right && __nondet()) {
            curr->right = malloc(sizeof(struct TreeNode));
            curr->right->left = NULL;
            curr->right->right = NULL;
        }
    }
    return root;
}

void destroyTree(struct TreeNode* root) {

```

```

while (root) {
    struct TreeNode* curr = root, * pred = NULL;
    while (curr->left || curr->right) {
        pred = curr;
        curr = (curr->left)?(curr->left):(curr->right);
    }
    if (pred) {
        if (curr == pred->left)
            pred->left = NULL;
        else
            pred->right = NULL;
    } else {
        root = NULL;
    }
    free(curr);
}

void dsw(struct TreeNode* root) {

    struct TreeNode sentinel;
    struct TreeNode* pred = &sentinel;
    struct TreeNode* succ = NULL;

    while (root != &sentinel) {

        succ = root->left;
        root->left = root->right;
        root->right = pred;
        pred = root;
        root = succ;

        if (root == NULL) {
            root = pred;
            pred = NULL;
        }
    }
}


int main(int argc, char* argv[]) {
    struct TreeNode* tree = createTree();
    dsw(tree);
    destroyTree(tree);
    return 0;
}

```

In the example we have 3 procedures, one creating the tree, one traversing the tree and one destroying the tree. Here we can verify that the pointer manipulations are correct (no null pointer dereferences for example) and that no garbage is created.

In general, we can verify safety properties of the considered programs which includes generic properties like absence of null dereferences, double free operations, dealing with dangling pointers, or memory leakage. Furthermore, to check various shape properties of the involved data structures one can use testers, i.e., parts of code which, in case some desired property is broken, lead the control flow to a designated error location. This is used for example to check that a doubly-linked list stays doubly-linked after a procedure manipulating it.

For the above purpose, we propose a novel approach of representing sets of heaps via *tree automata* (TA). In our representation, a heap is split in a canonical way into several *tree components* whose roots are the so-called *cut-points*. Cut-points are nodes pointed to by program variables or having several incoming edges. The tree components can refer to the roots of each other, and hence they are “separated” much like heaps described by formulae joined by the separating conjunction in separation logic [?]. Using this

	PROGRAMME ARPEGE EDITION 2009	VERIDYC DELIVERABLE 2
---	----------------------------------	--------------------------

decomposition, sets of heaps with a bounded number of cut-points are then represented by a new class of automata called *forest automata* (FA) that are basically tuples of TA accepting tuples of trees whose leaves can refer back to the roots of the trees. Moreover, we allow alphabets of FA to contain *nested FA*, leading to a *hierarchical encoding of heaps*, allowing us to represent even sets of heaps with an unbounded number of cut-points (e.g., sets of DLL). Intuitively, a nested FA can describe a part of a heap with a bounded number of cut-points (e.g., a DLL segment), and by using such an automaton as an alphabet symbol an unbounded number of times, heaps with an unbounded number of cut-points are described. Finally, since FA are not closed under union, we work with sets of forest automata, which are an analogy of disjunctive separation logic formulae.

As a nice theoretical feature of our representation, we can show that *inclusion* of sets of heaps represented by finite sets of non-nested FA (i.e., having a bounded number of cut-points) is decidable. This covers sets of complex structures like SLL with head/tail pointers. Moreover, we show how inclusion can be safely approximated for the case of nested FA. Further, C program statements manipulating pointers can be easily encoded as operations modifying FA. Consequently, the symbolic verification framework of *abstract regular tree model checking* [?], which comes with automatically refinable abstractions, can be applied.

The proposed approach brings the principle of *local heap manipulation* (i.e., dealing with separated parts of heaps) from separation logic into the world of automata. The motivation is to combine some advantages of using automata and separation logic. Automata provide higher generality and flexibility of the abstraction (see also below) and allow us to leverage the recent advances of efficient use of non-deterministic automata. The use of separation allows for a further increase in efficiency compared to a monolithic automata-based encoding proposed earlier [?].

We have implemented our approach in a prototype tool called *Forester* as a gcc plug-in (available at <http://www.fit.vutbr.cz/research/groups/verifit/tools/forester/>). In our current implementation, if nested FA are used, they are provided manually (similar to the use of pre-defined inductive predicates common in works on separation logic). However, we show that *Forester* can already successfully handle multiple interesting case studies, proving the proposed approach to be very promising. For example, the program above is verified in 0.7 sec.


## 4.2 Doubly-linked Lists with Integer Values

The team at LIAFA has also worked on the extension of the Celia tool to support the analysis of programs manipulating doubly linked lists (DLL) *storing integer data*. (Notice that the previous approach does not deal with data stored in linked structures.) Celia has been presented in details in the previous deliverable, so we focus here only on the extension for DLL.

The DLL-like data structures appear in the case study provided by EDF, Nagios<sup>4</sup>, a host/service/network monitoring program written in C and released under the GNU

---

4. [www.nagios.org](http://www.nagios.org)

	<p style="text-align: center;">PROGRAMME ARPEGE EDITION 2009</p>	<p style="text-align: center;">VERIDYC DELIVERABLE 2</p>
---	--	--

GPL. This program has more than 100 Kloc and defines several recursive data structures, mainly singly linked, doubly linked, and skip lists, as well as hash tables over these recursive data structures. We focus on the only DLL-like data structure of Nagios, called `timed_event`. The file defining the operations on this type has 1.7 Kloc and uses several features that has to be abstracted away in order to use Celia, mainly pointers to functions, arrays of untyped (`void*`) pointers, etc. The definition of this data structure and the declaration of the main functions which have been considered for the analysis are given in the listing below:


```
// Excerpt from file include/nagios.h
...
/* TIMED_EVENT structure */
typedef struct timed_event_struct{
    int event_type;
    time_t run_time; // long integer
    int recurring;
    unsigned long event_interval;
    int compensate_for_time_change;
    void *timing_func; // function pointer
    void *event_data; // array of untyped pointers for the function pointer above
    void *event_args; // array of untyped pointers for the function pointer above
    int event_options;
    struct timed_event_struct *next;
    struct timed_event_struct *prev;
}timed_event;

/* schedules a new timed event */
int schedule_new_event(int,int,time_t,int,unsigned long,void *,int,void *,void *,int);
/* reschedules an event */
void reschedule_event(timed_event *,timed_event **,timed_event **);
/* adds an event to the execution queue */
void add_event(timed_event *,timed_event **,timed_event **);
/* remove an event from the execution queue */
void remove_event(timed_event *,timed_event **,timed_event **);
/* resorts event list by event run time */
void resort_event_list(timed_event **,timed_event **);
```

The main idea of this work is to use an abstract domain to represent symbolically DLL data structures. This abstract domain has to be both (1) precise in order to keep the information about the length and data in the lists (e.g., sorting of data, data less than some limit for each list element) and (2) efficient for analysis of the programs with such data structures. The abstract domain used extends the existing one for singly linked lists (SLL) as follows. Values of this domain are sets of pairs combining a graph representation of the heap with a domain over data words which describes the data stored in the lists. The graph representation has been fully re-implemented to support DLL. The domain on the data words has been reused and extended with operations needed by the abstract transformers on DLL (i.e., operators implementing the effect of a statement on some list segment). Indeed, the data word domain of universal formulas can be used as for SLL if we consider that the universal positions are ordered using the `next` field. However, since a DLL segment may be split by a backward access using the `prev` field, we have added a new split operation in each data word domain. This operation separates the last element of the list from the beginning of the list.

Aside these main changes, we have had also to work on the extension of the interface between the abstract domain on shape graphs in order to support multiple pointer and data fields.

The analysis using Celia has been done after a simple but manual simplification of

	PROGRAMME ARPEGE EDITION 2009	VERIDYC DELIVERABLE 2
---	----------------------------------	--------------------------

the original code to remove unsupported features. The analysis have been done in a modular way, for each function in the listing above. Celia has been able to infer invariant properties for each function above. For example, for the `add_event` function is inferred the property that the length of the list has been increased by 1; moreover, if the pre-condition of `add_event` is that the input list is sorted, Celia infers that the output list is also sorted. The last property is obtained when analyzing the `resort_event_list` function, which calls the `add_event` function to do insertion sort wrt the `run_time` field.

### 4.3 Shallow Pointers and Pointer Arithmetic

During the second year of the project, the VERIMAG team has been working on verification of C programs with low-level pointer arithmetic operations, under the assumption that the program does not use recursively defined data structures (lists, trees, etc.), hence the name of *shallow pointers*. Despite the lack of recursive data structures, such programs are commonplace in industrial control software, which is the case of the examples supplied by the EDF partner within the project. An example of an industrial application of shallow pointers is the rotating buffer test case, presented in Section 5.

The main idea is to reduce the verification of such programs to decision problems on hierarchical *Numerical Transition Systems* (NTS) [?] – essentially non-deterministic integer programs with function calls. This translation is enabled by a static analysis phase, in which the shape of the memory at each control point is computed approximately. The symbolic representation used here is an in-house dialect of Separation Logic [?], described in the following. The static analysis is modular, i.e. it is executed for each procedure, in isolation, and infers the most general Hoare triple (pre- and post-condition) under which the function may execute without a memory fault (null pointer dereference or memory leak).

For instance, the Separation Logic formula  $\exists \ell_1 \exists \ell_2. x \rightarrow \ell_1 \wedge y \rightarrow \ell_2 \mid alloc(\ell_1) * alloc(\ell_2)$  states the existence of two different allocated memory cells  $\ell_1$  and  $\ell_2$ , pointed by the program variables  $x$  and  $y$ , respectively. The analysis propagates such formulae forward, by computing postconditions as in e.g.:

$$\left\{ \begin{array}{l} \{\exists \ell_1 \exists \ell_2. x \rightarrow \ell_1 \wedge y \rightarrow \ell_2 \mid alloc(\ell_1) * alloc(\ell_2)\} \\ \quad \mathbf{free}(x) \\ \{\exists \ell_1 \exists \ell_2. x \rightarrow \ell_1 \wedge y \rightarrow \ell_2 \mid alloc(\ell_1)\} \\ \quad \mathbf{free}(y) \\ \{\exists \ell_1 \exists \ell_2. x \rightarrow \ell_1 \wedge y \rightarrow \ell_2 \mid emp\} \end{array} \right.$$

This analysis can detect several memory faults, such as null pointer dereference, or double free. Even if the static analysis phase is successful, this however, does not guarantee program correctness, since the symbolic model used is too coarse to represent pointer arithmetic. For this reason, we generate an NTS that captures pointer arithmetic, and use an existing off-the-shelf verification tool for NTS (e.g. FLATA[?] or ELGARICA[?]), in order to detect finer errors, such as, e.g. unaligned memory accesses, freeing a pointer not pointing to the beginning of an allocated area, etc.

### 4.3.1 Example

Below we show a small C program (left), annotated with Separation Logic invariants, together with the NTS generated by the FLATA-C [?] plugin (right), developed at VERIMAG. The output of the tool is basically an automaton augmented with registers (counters) which are updated by first-order formulae, describing linear arithmetic relations. One inherent difficulty, when using a low-level logical formalism is the need to explicitly specify the *frame conditions* (the set of counters which are not affected by the transformation). This explains the extensive use of the `havoc` primitive (implicit copy of all counters not listed as arguments).

```

main(){
  int i,j, *x;

  {emp}

  x=(int *)malloc(100*sizeof(int));
  {∃ℓ.x → ℓ | alloc(ℓ)}

  i = 0;
  j = NonDetInt();
  {∃ℓ.x → ℓ | alloc(ℓ)}

  while(*(x+i) != 0)
  {∃ℓ.x → ℓ | alloc(ℓ)}
    i++;
}

```

```

main{ s0->s1 { 100*4 <= 0 && havoc() }
s0->s2 { offset__x_:=0 && 100*4 > 0
      && mid_1_size:=100*4 && mid_1_base:=1 &&
      havoc(mid_1_base,mid_1_size,offset__x_) }
s1->s4 { i'=0 && havoc(i) }
s2->s5 { i'=0 && havoc(i) }
s4->sinter0 { havoc() }
sinter0->s7 { (j',validity__j_)=NonDetInt() &&
      havoc(j) }
s5->sinter1 { havoc() }
sinter1->s8 { j'=NonDetInt() &&
      havoc(j) }

s7->s9 { havoc() }
s8->s10 { havoc() }
s9->s11 { havoc() }
s10->s12 { havoc() }
s11->s13 { i'=i+1 && havoc(i) }
s12->s13 { not ((offset__x_+i*4 < mid_1_size &&
      offset__x_+i*4 >= 0) &&
      (offset__x_+i*4)\%4 = 0) && i'=i+1
      && havoc(i) }
s12->s16 { __if_ndet_cond__ = 0 &&
      ((offset__x_+i*4 < mid_1_size &&
      offset__x_+i*4 >= 0) &&
      (offset__x_+i*4)\%4 = 0) &&
      i'=i+1 && havoc(i) }
s12->s17 { __if_ndet_cond__ != 0 &&
      ((offset__x_+i*4 < mid_1_size &&
      offset__x_+i*4 >= 0) &&
      (offset__x_+i*4)\%4 = 0) && havoc() }

s14->s9 { havoc() }
s15->s18 { havoc() }
s16->s10 { havoc() }
s17->s19 { havoc() }
s18->s20 { j >= 0 && havoc() }
s18->s22 { j < 0 && tmp_0:=0 && havoc(tmp_0) }
s19->s23 { j >= 0 && havoc() }
s19->s24 { j < 0 && tmp_0:=0 && havoc(tmp_0) }
s20->s26 { j < i && tmp_0:=1 && havoc(tmp_0) }
s20->s28 { j >= i && tmp_0:=0 && havoc(tmp_0) }
s22->s30 { __retres:=1 && havoc(__retres) }
s23->s31 { j < i && tmp_0:=1 && havoc(tmp_0) }
s23->s32 { j >= i && tmp_0:=0 && havoc(tmp_0) }
s24->s33 { __retres:=1 && havoc(__retres) }
s26->s30 { __retres:=1 && havoc(__retres) }
s28->s30 { __retres:=1 && havoc(__retres) }
s30->s34 { ret_val_:=__retres && havoc(ret_val_) }
s31->s33 { __retres:=1 && havoc(__retres) }
s32->s33 { __retres:=1 && havoc(__retres) }
s33->s35 { ret_val_:=__retres && havoc(ret_val_) } }

```



### 4.3.2 Distribution and Usage of FLATA-C

FLATA-C is distributed freely, under LGPL public license on the website <http://www-verimag.imag>. To use Flata-C one must first download and install Frama-C. The tool can be invoked currently by the command line: `frama-c flatac <filename.c>`

## 5 Industrial Test Cases

### 5.1 Rotating Buffer

The example in Figure 2 is inspired by the code of an application supplied by EDF. The two functions `put` and `get` provide access to a buffer data structure used by several reader and writer threads. The buffer data structure is given below:

```
typedef struct {
    char* head;
    char* out;
    char* in;
    int size;
    int free;
    int count;
} buffer_t;
```

This is a typical example of a shallow pointer data structure (notice the lack of recursion in the above type definition), which is updated using pointer arithmetic operations. We currently analyse this example with the FLATA-C plugin tool.

### 5.2 MTHREAD Test Case

We have used the MTHREAD plugin on the EDF-supplied code from which the rotating buffer code is extracted. This work is preliminary: the case study is partially out of the scope of the Value Analysis, mainly because of the presence of dynamic allocation, and requires some selective rewriting to remove the more problematic cases of `malloc`.

The current results of the shared memory analysis are as follows:

```
[bufLLC.bufHead] write unprotected, read protected by (?)&bufLLC+24
[bufLLC{.currentIn; .currentOut; }] write protected by (?)&bufLLC+24,
    read protected by &bufLLC+24
[bufLLC.size] write unprotected, read protected by &bufLLC+24
[bufLLC.free] write protected by (?)&bufLLC+24, read protected by &bufLLC+24
[bufLLC.frameNb] protected by &bufLLC+24

[bufTSDU.bufHead] write unprotected, read protected by (?)&bufTSDU+24
[bufTPDU.bufHead] write unprotected, read protected by (?)&bufTPDU+24

[codeEventApid] unprotected
[codeEvent] unprotected
[codeEventAna] unprotected
[codeEventPilote] unprotected
```

The variables `codeEvent*` are detected as being accessed unprotected. A code review confirms this fact, but this does not necessarily indicates a defect in the code. Indeed,

```

int put(buffer_t* b, int size, char* data) {
    // check for space
    if (b->free < 4 + size) {
        trace("put: aborted, overflow");
        return KO_OVERFLOW;
    }
    // write the length on 4 bytes
    if (b->out + 4 <= b->head + b->size) {
        * ((int*)(b->out)) = size;
        b->out += 4;
        b->free -= 4;
        // cycle, if needed
        if (b->out == b->head + b->size)
            b->out = b->head;
    }
    // write the data on 'size' bytes
    if (b->out + size <= b->head + b->size) {
        // data fits entirely
        memcpy(b->out, data, size);
        b->out += size;
    }
    else {
        // data needs to be 'fractioned' in two
        int s = (b->head + b->size) - b->out;
        memcpy(b->out, data, s);
        memcpy(b->head, data + s, size - s);
        b->out = b->head + size - s;
    }
    // update for free space
    b->free -= size;
    b->count++;
    // re-align on 4 bytes
    if (size % 4) {
        b->out += (4 - size % 4);
        b->free -= (4 - size % 4);
    }
    // cycle, if needed
    if (b->out == b->head + b->size)
        b->out = b->head;
    return OK;
}

int get(buffer_t* b, int* size, char** data) {
    if (b->count == 0) {
        trace("get: aborted, underflow");
        return KO_UNDERFLOW;
    }
    // read the data size
    *size = * ((int* ) b->in);
    b->in += 4;
    b->free += 4;
    // cycle, if needed
    if (b->in == b->head + b->size)
        b->in = b->head;
    // allocate memory and read the data
    *data = malloc( *size );
    if (b->in + (*size) <= b->head + b->size) {
        // data comes in one reading
        memcpy(*data, b->in, *size);
        b->in += *size;
    }
    else {
        // data is split,
        int s = (b->head + b->size) - b->in;
        memcpy( *data, b->in, s);
        memcpy( *data + s, b->head, *size - s);
        b->in = b->head + (*size - s);
    }
    // update for free space
    b->free += *size;
    b->count--;
    // realign on 4 bytes
    if ( *size % 4 ) {
        b->in += (4 - *size % 4);
        b->free += (4 - *size % 4);
    }
    // cycle, if needed
    if (b->in == b->head + b->size)
        b->in = b->head;
    return OK;
}

```

Figure 2: Rotating buffer access functions

those variables are scalar values, that are used as monotonic (increasing) bit-masks of errors. On standard architectures, protecting those accesses is indeed not needed.

The buffers `bufLLC`, `bufTSDU` and `bufTPDU` are partially protected by their respective mutexes, contained at offset 24 in the structure. In some cases, the accesses are detected as not protected. Using the verbose output, which displays all the instructions that perform the accesses (see §2.3), we notice that those are done by the main thread. More precisely, they occur in a function that stops the program, the call-stack being `Buf_Free <- Stop_Analyseur`. It is possible that those accesses occur only after the other competing threads have been stopped. Since we do not currently take condition variables into account, this is not automatically detected by `MTHREAD`. Still, the user already has very precise information on what must be checked to ensure proper lock discipline.

Finally, Figure 3 presents the concurrent control-flow graphs of two threads of the program. Blue nodes are concurrent accesses to a shared variable, while green ones are

Figure 3: Examples of MTHREAD concurrent control-flow graphs

*non*-concurrent accesses to a shared resource (the other competing threads are either suspended or not created yet). Red nodes represent thread creation or cancellation (the topmost red nodes in the left graph), or mutex locking or release.

While the resolution is not sufficient to read the labels on the nodes, the shapes of the graphs are interesting in their own right. The leftmost graph is the CCFG of the main thread, while the rightmost one is a synchronization thread. Unsurprisingly, the shapes of the two graphs are completely different. Some typical code structures are nevertheless present, and are very apparent on our simplified graphs:

- a sequence of thread creations, at the top of the main thread;
- a sequence of thread cancellations, on the left of the main thread;
- two infinite loops (one in each thread) materialized by return edges on the `while(1)` node;
- multiple concurrent accesses in the body of the infinite loop, protected by a mutex (synchronization thread).

## 6 Ongoing and Future Work

The VERIDYC project has entered its final phase, in which concurrent C code with complex data structures is considered. This phase has been prepared, on one hand by considering complex data structures in the sequential setting (the FLATA-C and CELIA tools), and on the other hand, by considering concurrent programs with singly-linked lists data structures (the HEAPHOP tool). Moreover, several concurrency models are analysed using an interleaving-free parallel programming model, and the complexity of their verification problems has been studied.

Future work includes tighter tool integration, in particular with the MTHREAD plugin for concurrent C code analysis, as well as implementing a verification method for concurrent programs that may handle complex data structures. We plan on carrying out experiments using industrial test cases, such as the rotating buffer example provided by EDF.