Model Checking C Programs with Dynamic Memory Allocation[*]

María del Mar Gallardo, Pedro Merino, and David Sánan
Department of Computer Science, University of Málaga, Spain
Emails: {gallardo,merino,sanan}@lcc.uma.es

Abstract

Software model checking technology is based on an exhaustive and efficient simulation of all possible execution paths in concurrent programs. Existing tools based on this method can rapidly detect execution errors, preventing malfunctions in the final system. However dealing with dynamic memory allocation is still an open trend.

In this paper, we present a novel method to extend explicit model checking of C programs with dynamic memory management. The method consists in defining a canonical representation of the heap that is based on moving most of the information from the state vector to a global structure. We give a formal semantics of the method in order to show its soundness. Our experimental results show that this method can be efficiently implemented in many well known model checkers, like CADP *or* SPIN.

Keywords: *Model extraction, software model checking, pointers, dynamic memory*

1 Introduction

Model checking is a mature technique to analyze properties of concurrent and critical systems, both hardware and software. Research in this topic has produced a number of tools oriented to particular specification languages, usually called *formal description techniques*. During the last ten years, this method has been adapted to real programming languages, like C[11] [1] [16] [5], C++[8] or JAVA[9] [3] [13]. These tools for *software model checking* are based on the same state exploration algorithms designed for formal description techniques, and not all of them do consider additional features missing in specification languages, like pointers and dynamic memory allocation.

1.1 Motivation and related work

Extending explicit model checking to manage programming languages with dynamic memory allocation presents two kinds of problems: the internal representation of dynamic structures (the *heap*) and the specification/verification of properties related to these dynamic structures. In the following paragraphs, we describe works related to these two areas.

1.1.1 Heap representation

From the implementation point of view, the problem is how to deal with the internal representation of the states during the exploration of the potential behaviors of the program. Model checking algorithms are optimized to consider global states with a fixed structure and length and they should be modified in order to deal with states with different configurations that depend on operations to allocate and free memory. There are some proposals describing representations of the state for C [6] [16] and JAVA [13].

The most natural approach to deal with dynamic structures is to allocate a heap for every process in the state vector. In that way, the state vector contains all the static and dynamic variables for all the processes, although a high cost on memory use has to be paid. This is the approach initially followed by the tool CMC [16], which is able to do model checking of C and C++ programs. In order to avoid the huge use of memory, CMC employs a hash table where only a signature of the state is stored, so this kind of compression of the state vector produces partial verification.

Tool dSPIN [6] extends SPIN with new PROMELA sentences and modifies the basic SPIN implementation. The language is extended with a notation to identify pointer variables, in such a way that the operations regarding pointers (assignment and comparison) are given a different semantics which is context-dependent. Their behavior depends on the position of the pointer variable (left or right) and the type of the other side of the instruction. Internally, the tool uses an extensible vector state with a separate area for dynamic objects for each state. This extensible state is linearized in every step of the model checker in order to pro-

[*]This work has been partially supported by the Spanish MEC under grant P07-TIC3131, TIN2007-67134 and IST-5-033563

duce a representation compatible with the SPIN algorithms to perform matching, hashing and state compression. Apart from considering the heap in every state (at least partially), this linearization is a time consuming step; however, only the relevant information should be copied to the linear state using a canonical representation of the heap. This idea is also applied in BOGOR [18].

JPF[13], a Java oriented model checker, also considers the separation of static and dynamic parts of the state vector. Dynamic objects are stored as a global pool of values, and only the indexes to the pool are placed in the static part of the state vector, together with the static variables. This way of collapsing the state increases the time and memory to verify large examples due to backtracking (necessary to perform exhaustive exploration of the bytecode corresponding to the Java program). So the authors also implement a reverse collapse method to manage the states.

1.1.2 Verifying properties over dynamic structures

It is necessary to define new property languages to express requirements and to reason about data structures created in a dynamic form, like linked lists. Model checkers usually employ variants of temporal logic to define properties about states and sequences of states. Atomic propositions in these logics are related to the static variables in the program. When considering linked structures (with anonymous nodes), a new mechanism is needed to reason about them.

Tool GROOVE [12] focuses on making the dynamic structures available to check CTL formulas. Instead of a linear method, the authors use a graph representation, which is more suitable to implement efficient matching.

The proposal by Bouajjani et al. [2] focuses on the use of a global store of values to represent linked structures as list and graphs. Then they employ model checking combined with abstraction to reason about the dynamic structures.

It is also worth noting that some works focus on designing new logics to deal with dynamic structures rather than on the internal behaviour of model checkers. In particular, work has been done in separation logic [17] parametric shape analysis [1] and pointer assertion logic [15]. However, these logics are used in theorem provers and its aplicability in model checking techniques have to be studied.

1.2 Contributions

The problem addressed in this paper is how to represent the state vector during verification of C programs with dynamic structures. The contributions of the paper are: a) a novel method to deal with pointers and heap management in the context of C programs; b) the formalization of the method in order to prove its correctness; and c) an extension of the tool set CADP to include this functionality.

We propose a new representation of the heap of a given C process that consists in using an incremental global data structure to allocate new objects. This global store is not kept as a part of the state; instead, indexes to this store are only used to point to the store elements. The way we generate the indexes (a hashing method) and the way of managing the store allows us to efficiently manage canonical representation of the states and to implement model checkers with this feature. This approach constitutes a new way to implement state collapsing [10].

This new proposal to manage pointers does not affect the expected behavior of model checkers when analyzing programs without pointers. We can use the usual model checker property language, like temporal logic, to reason about programs that use pointers and dynamic memory; however, we still cannot reason about the dynamic structures itself. This feature will be our next work.

Previous works cited above only provided informal descriptions of the mechanism to manage dynamic memory. We describe our proposal giving a formal semantics to the operations related to pointers and memory management. As far as we know, this is the first time that collapsing related methods are formalized. Like in [4], the formal semantics has been useful to check correctness and to guide the implementation.

Our proposal can be implemented in many existing tools. The first implementation has been carried out within our project C.OPEN [5], which extends the tool box CADP in order to verify C programs that use external functions. Experimental results confirm that we can use the extension with realistic C programs.

Compared to related work, our method shares the ideas of the global store and canonical states with [13] and [2]. However, like JPF our way of managing the store is more efficient, because we save memory keeping the real data outside the state vector. We also save time because: a) we do no need the linearization used in dSPIN and b) we do not need a specific mechanism to implement backtracking. This is due to the use of a special hashing method to manage the global store. Furthermore, our method considers new features such as explicit memory deallocation and support for pointer arithmetic.

The paper is organized as follows. In Section 1, we present a summary of our approach to integrate heap management in the usual algorithms to perform explicit model checking. The approach is formalized and analyzed for soundness in Section 3. Sections 4 and 5 give details on implementation issues and experimental results, respectively. Conclusions are given in Section 6

2 Modelling the heap of a C process

Explicit model checking (of software) consists in generating all the global states of a given configuration of concurrent programs. The most resource consuming operation is checking whether a given (new) state has been analyzed before. The number and size of global states to be compared for this purpose depends on the variables in the processes and on the degree of interleaving. So usually, optimizations are considered in both directions, reducing both the representation of states and the interleavings necessary to check a property. However, there is a balance between the optimizations and the visibility of instructions and variables needed to check the properties. This is also applicable when we extend model checking to deal with dynamic memory management in C programs.

In theory, dynamic memory can produce an infinite number of states and, therefore, explicit model checking does not work to check properties. However, if we consider realistic programs, we could think of practical methods to manage very large states. The critical point is how to keep a small representation of states while capturing the relevant information to compare states. Our method produces such a canonical representation. Figure 1 represents how we split the information of the states into two parts. The state vector contains the relevant information to be used during model checking, for instance, to compare states and to check properties. The global store contains all the data structures created by processes at any point during model checking. We add any new value created to the store, thus ensuring that these values are shared as parts of the heaps of different processes (this allows us to keep the store in a reasonable size). The heap of every process is placed in the state, but only a reference to the store is put in the process heap.

The logical model of the process heap works as follows. Pointer variables are usually static process variables, therefore they are placed with other variables. A pointer variable contains a reference to the process heap, which is uniquely assigned depending on the name of the variable (this is done with a hash function). The entry in the heap contains a reference to the value in the store (or a Null reference). Each reference is composed of the type of the dynamic structure plus a hash value that is fixed for each value of the given type during model checking. This second hash function is a critical part that clearly depends on the range of values allowed for every type (types can be C native or user defined). Keeping each process heap sorted by the values produced by the first hash function (the one based on the names), we obtain a canonical representation of the states. This canonical representation ensures that different interleavings reaching the same state will be correctly considered. All these mechanisms allow us to ignore the global store when comparing states. The information in the store is only used to produce new states.

This heap model allows us to easily represent non-linked structures. The model is also able to manage linked structures by including references to the heap in the global store. For instance, figure 1 shows how a list is internally represented. The list is pointed to by the variable `list`, which refers to an element in the heap containing the information about the location in the store. The next element in the list is pointed by the field `next`, which has the reference to the dynamic area in the heap (or `NULL` if it has not been linked to the following element). Fortunately, we can again ignore these references in the store when comparing two states, because we have all the information in the process heap.

Regarding the instructions to deal with pointers, we explicitly capture and model the creation and deallocation of memory and all the operations to access the objects. The way of modelling these operations and the overall method to manage dynamic memory isformalized and analyzed for correctness in the following section.

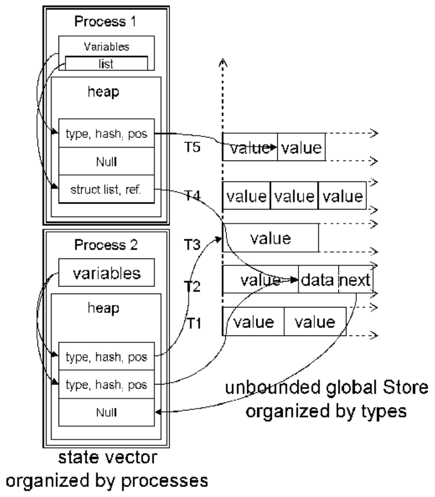

Figure 1. Model of the heap

3 Formal Description

In this section, we give a formal description of the proposal to deal with dynamic memory. To simplify the presentation, we have assumed that our program consists of a single process. Since our concurrency model for C only supports local variables, this assumption is not a real restriction. The extension to systems with more than one process may be obtained by replicating the model described here in the natural way.

3.1 Memory Model

Let $\mathcal{V}$ be the set of variables declared in a given C program P. We assume that variables in $\mathcal{V}$ lexicographically ordered. Thus, if $x \in \mathcal{V}$, δ_x denotes the position of x in $\mathcal{V}$. Let us denote with $Value$ the set of all possible values of variables in P. We assume that $Value$ contains the set of natural numbers $\mathbb{N}$ and the set $\mathcal{A}$ of addresses that may point to dynamic data during execution. As it was described in the previous section, in our model, the memory handled by P is divided into three main parts $\gamma = \langle \gamma_s, \gamma_h, \gamma_{\text{ST}} \rangle$ where $\gamma_s \in \text{STATIC} = \mathcal{V} \rightarrow Value$ is the static part of state, with the current value of each variable declared in P. If ptr is a pointer variable, then $\gamma_s(ptr) \in \mathcal{A}$. $\gamma_{\text{ST}} \in \text{STORE} = \mathcal{T} \times \mathbb{N} \rightarrow Value^*$ where $\mathcal{T}$ is the set of type identifiers used in the program and $Value^*$ denotes the set of all sequences $v_1 \cdots v_n$ of any length $n \geq 0$ of values. γ_{ST} is the global structure that stores all the dynamic data created during execution. Each value $v \in Value$ of type t is saved in the sequence $\gamma_{\text{ST}}(t, \text{HASH}(v))$, $\text{HASH} : Value \rightarrow \mathbb{N}$ being a proper hash function. Although store γ_{ST} is included in γ, we will see later (in Section 3.4) that the way of handling γ_{ST} allows us to consider it as a global external structure. Finally, component γ_h in γ codifies the memory heap. Formally, $\gamma_h \in \text{HEAP} = \mathcal{A} \rightarrow \mathcal{T} \times \mathbb{N} \times \mathbb{N} \cup \{null\}$ is a function that associates addresses with a triples $(type, hash, pos)$ which give us the position of the corresponding dynamic variable on the global store γ_{ST}. Value $null$ indicates that the corresponding address is not being used. Thus, as explained above, given a pointer variable ptr, $\gamma_s(ptr)$ gives us the current address of ptr in the heap, and $\gamma_h(\gamma_s(ptr))$ provides the position of the dynamic variable pointed by ptr at the global store. Thus, $\gamma_h(\gamma_s(ptr)) = (t, n, m)$ tell us that this variable is the m-th value of sequence $\gamma_{\text{ST}}(t, n) = v_1 \cdots v_k$, that is, v_m $(m \leq k)$. In the sequel, *configurations* are the 3-tuples $\gamma = \langle \gamma_s, \gamma_h, \gamma_{\text{ST}} \rangle$ that define the program state at a point during execution. Let us denote with $\mathcal{C}$ the set of configurations.

3.2 Notation

The following section gives semantics to the pointer operations. We now describe the notation used in the description of rules given below.

3.2.1 Sequences

Given $seq = v_1 \cdots v_n \in Value^*$ and $val \in Value$, $seq \cdot val$ denotes the sequence $v_1 \cdots v_n \cdot val$ obtained appending val to the end of seq. Operator "$\in$" is used to check whether a given value val is in a sequence . That is, $c \in v_1 \cdots v_k$ iff $\exists m \leq k . v_m = c$. Given a sequence $seq = v_1 \cdots v_k$, $seq \downarrow m$ denotes the m-th value of seq, that is, v_m. We use operator $+$ to append a value val to a sequence seq as follows.

$$seq + val = \begin{cases} seq, \textit{if } val \in seq \\ seq \cdot val, \textit{if } val \notin seq \end{cases}$$

That is, val is only appended if it does not belong to seq. Given a sequence seq of distinct values and $val \in Value$, function $ind : Value \times Value^* \rightarrow \mathbb{N}$ returns the position of a value in a given sequence, that is, $ind(val, seq) = m$ iff $seq \downarrow m = val$, and $ind(val, seq) = 0$, otherwise. Finally, we denote with $len(seq)$ the length of a sequence.

3.2.2 Pointer Codification

As commented above, HEAP makes use of codified addresses to relate pointers with the dynamic data stored in the global store. To do this, we assume that for each variable $x \in \mathcal{V}$, there exists an infinite sequence of distinct addresses $x_1 \cdot x_2 \cdots$ to be used as contents of variable x during execution. In addition, addresses used by different variables are also distinct. Thus, we assume that $cod : \mathcal{V} \rightarrow \mathcal{A}$ is a codification function that returns a distinct number (a codification) for each variable x. In addition, implementation of cod guarantees that $cod(x) = x_i$ iff $cod(x)$ has been called exactly i times.

3.2.3 Evaluation of Expressions

In the following rules, we assume that $\mathcal{E}$ is the set of program expressions and that function $eval : \mathcal{E} \times \mathcal{C} \rightarrow Value$ returns the result of evaluating each expression in a given configuration. Function $eval$ may make use of the three components to calculate the returned value as follows:

- $eval(x, \gamma) = \gamma_s(x)$, if $x \in \mathcal{V}$
- $eval(\&x, \gamma) = \delta_x$, if $x \in \mathcal{V}$
- $eval(*ptr, \gamma) = v_m$ iff $ptr \in \mathcal{V}$ is a pointer variable such that $\gamma_h(\gamma_s(ptr)) = (t, n, m)$ and $\gamma_{\text{ST}}(t, n) \downarrow m = v_m$.
- $eval(ptr \rightarrow c, \gamma) = v_m \cdot c$ iff $ptr \in \mathcal{V}$ is a pointer variable such that $\gamma_h(\gamma_s(ptr)) = (t, n, m)$ and $\gamma_{\text{ST}}(t, n) \downarrow m = v_m$.
- $eval(\&(ptr \rightarrow c), \gamma) = \&(v_m \cdot c)$ iff $ptr \in \mathcal{V}$ is a pointer variable such that $\gamma_h(\gamma_s(ptr)) = (t, n, m)$ and $\gamma_{\text{ST}}(t, n) \downarrow m = v_m$.

Above, $v_m \cdot c$ denotes the field c of record v_m, and $\&(v_m \cdot c)$ represents the physical address of $v_m \cdot c$ in the state.

3.3 Semantics Rules

We now define the semantics of the C instructions related to dynamic memory management (denoted by Inst) as a labelled transition system (LTS) $\langle \mathit{Conf}, \xrightarrow{\mathit{Inst}} \rangle$, Conf being the set of configurations. Below, for each function f, $f[\ x \ :\ v\ ]$ denotes the function that is equal to f for all elements in its domain except for x that is associated to v.

3.3.1 Allocation

If $x \in \mathcal{V}$ and $cod(x) = x_i$ then
$$\gamma \xrightarrow{x = malloc(sizeof(t))} \langle \gamma_s[x : x_i], \gamma_h[x_i : (t,0,0)], \gamma_{\text{ST}} \rangle$$
That is, to create a new dynamic data of type t, we update the configuration, binding the new reference x_i to the static variable x in γ_s, and giving value $(t,0,0)$ to address x_i in γ_h. Value $(t,0,0)$ means that the dynamic value pointed by x has been created but it has not yet been initialized. That is why component γ_{ST} is not modified.

3.3.2 Deallocation

To deallocate a pointer variable, we only eliminate the reference to the global store in γ_h.

$$\gamma \xrightarrow{free(ptr)} \langle\ \gamma_s, \gamma_h[\gamma_s(ptr) : null], \gamma_{\text{ST}} \rangle$$

3.3.3 Assignment

Case x = val $\gamma \xrightarrow{x=val} \langle \gamma_s[x : eval(val,\gamma)], \gamma_h, \gamma_{\text{ST}} \rangle$

When an assignment is executed on a static variable, only the static part γ_s is modified.

Case ptr $\rightarrow$ c = val For each type $t \in \mathcal{T}$, we denote with ϵ_t the value of type t with all its fields initialized, and with $\epsilon_t[c : v]$ the value of type t with all its fields initialized except c that stores v.

1. If $\gamma_h(\gamma_s(ptr)) = null$ then
$$\gamma \xrightarrow{ptr \rightarrow c = val} error$$
That is, if ptr has not been previously allocated or has been freed, instruction $ptr \rightarrow c$ produces an execution error.

2. Assume that $\gamma_h(\gamma_s(ptr)) = (t,0,0)$, that is, the dynamic variable has not yet initialized. Let $w = \epsilon_t[c : eval(val,\gamma)]$, then if $\text{HASH}(w) = h$ and $ind(w, \gamma_{\text{ST}}(t,h) + w) = m$
$$\gamma \xrightarrow{ptr \rightarrow c = val} \langle\ \gamma_s, \gamma_h[\gamma_s(ptr) : (t,h,m)], \gamma_{\text{ST}}[(t,h) : \gamma_{\text{ST}}(t,h) + w] \rangle$$
The new dynamic data $w = \epsilon_t[c : eval(val,\gamma)]$ is added, using operator $+$, in the proper position in the sequence of values, and the heap is updated with the position of the data in the store. Observe that this rule models both the case when the new data w is already in the sequence (and the sequence is not modified), and the case when w does not appear in the sequence (and it is added at the end).

3. Assume now that ptr points to a dynamic variable stored in γ_{ST}, that is, $\gamma_h(\gamma_s(ptr)) = (t,h,m)$ with $h \neq 0$ and $m \neq 0$. Let w denote the new data $w = \gamma_{\text{ST}}(t,h) \downarrow m[c : eval(val,\gamma)]$ to be introduced in the store. Observe that w is constructed by substituting the c field of the current dynamic data pointed by ptr, by the new value $eval(val,\gamma)$. If $\text{HASH}(w) = h'$ and $ind(w, \gamma_{\text{ST}}(t,h') + w) = m'$, then
$$\gamma \xrightarrow{prt \rightarrow c = val} \langle\ \gamma_s, \gamma_h[\gamma_s(ptr) : (t,h',m')], \gamma_{\text{ST}}[(t,h') : \gamma_{\text{ST}}(t,h') + w] \rangle$$
As before, due to the operator $+$ definition, this rule takes into account the cases when $w \in \gamma_{\text{ST}}(t,h')$ and $w \notin \gamma_{\text{ST}}(t,h')$

Case $*$ptr = val The cases for this instruction are similar to the previous ones, except that it is not necessary to refer to any field to access the element pointed by ptr to be modified.

We assume that the program to be analyzed only uses the instructions with pointers given above. This is not a real restriction, because sentences involving more complicated pointer dereferences may be written as a sequence of these instructions. For example, `ptr → next = malloc(sizeof(t))` may be written as `temp1 = malloc(sizeof(t)); ptr → next = temp1`, and `ptr → next → next = exp` may be translated into the sequence `temp1 = ptr → next; temp1 → next = exp`.

Our model supports the casting C feature as the rest of (non pointer-related) C operations. The expression containing the casting is isolated and its value is calculated by the underlying C runtime environment.

3.4 Correctness

This section is devoted to proving that the memory model and the behavior defined by rules described above allows us to implement a *simplified* matching function in the model checking algorithm. In particular, the way of dealing with pointers and dynamic data in our model makes it unnecessary to check the global store (we only take into account components γ_s and γ_h), which entails a considerable improvement in model checking performance.

To do this, we now introduce a notion of *equivalence* of configurations that captures the intuitive idea that two configurations match if they have the same values for the static and the dynamic program data. We use *renaming* functions to define this notion. Let $\rho : \mathcal{A} \rightarrow \mathcal{A}$ be a one-to-one function that renames references. ρ may be extended to values ($\rho : Value \rightarrow Value$) in the natural way, that is, preserving all values except addresses that are renamed via ρ. Given a configuration $\gamma = \langle \gamma_s, \gamma_h, \gamma_{\text{ST}} \rangle \in \mathcal{C}$ and a renaming function $\rho : \mathcal{A} \rightarrow \mathcal{A}$, we may construct the renamed configuration $\gamma^\rho = \langle \gamma_s^\rho, \gamma_h^\rho, \gamma_{\text{ST}}^\rho \rangle$ where:

1. $\gamma_s^\rho(v) = \rho(\gamma_s(v)), \forall v \in \mathcal{V}$.
2. $\gamma_h^\rho(d) = \gamma_h(\rho^{-1}(d)), \forall d \in \mathcal{A}$.
3. $\gamma_{\text{ST}}^\rho(t,h) \downarrow m = \rho(\gamma_{\text{ST}}(t,h) \downarrow m), \forall t \in \mathcal{T}, h, m \in \mathbb{N}$ such that $m \leq len(\gamma_{\text{ST}}(t,h))$.

Intuitively, the renamed configuration γ^ρ is equal to γ, except for the references given to pointer variables that have been changed using ρ. That is, both γ and γ^ρ define the *same* memory state modulo the references used to store data.

Definition 1 *Let $\gamma, \gamma' \in \mathcal{C}$ be two configurations, we say that γ and γ' are equivalent ($\gamma \cong \gamma'$) iff there exists a renaming function $\rho : \mathcal{A} \rightarrow \mathcal{A}$ such that $\gamma_s^\rho = \gamma'_s$, and $\forall d \in \mathcal{A}$ one of the following conditions holds:*

1. *$\gamma_h^\rho(d) = null$ and $\gamma'_h(d) = null$.*
2. *$\gamma_h^\rho(d) = \langle t, h, m \rangle, \gamma'_h(d) = \langle t, h', m' \rangle$, and $\gamma_{\text{ST}}^\rho(t,h) \downarrow m = \gamma'_{\text{ST}}(t,h') \downarrow m'$.*

That is, two configurations γ and γ' are equivalent if there exists a renamed configuration γ^ρ such that γ^ρ and γ' give the same values to all (static and dynamic) program variables. In the sequel, we prove that the task of checking the equivalence of configurations may be strongly simplified when dealing with the memory model described above. We first introduce a partial order $\sqsubseteq$ over stores:

Definition 2 *Let $\Gamma, \Gamma' \in$ STORE be two stores, $\Gamma \sqsubseteq \Gamma'$ iff $\forall t \in \mathcal{T}, h, m \in \mathbb{N}$ such that $m \leq len(\Gamma(t,h))$, it holds that $m \leq len(\Gamma'(t,h))$ and $\Gamma(t,h) \downarrow m = \Gamma'(t,h) \downarrow m$.*

That is, $\Gamma \sqsubseteq \Gamma'$ iff Γ' may be obtained adding new dynamic data at the end of the sequences of values of Γ.

Proposition 1 *Let $\gamma, \gamma' \in \mathcal{C}$ be two configurations, and $cinst \in Inst$ a C instruction such that $\gamma \xrightarrow{cinst} \gamma'$, then $\gamma_{\text{ST}} \sqsubseteq \gamma'_{\text{ST}}$.*

Proof 1 *To prove the result, we should only observe that the C instructions that modify the global store γ_{ST} are those related to pointers. In addition, rules described in Section 3.3 never eliminate data from the store. On the contrary, when store γ_{ST} is modified, it is extended with new elements at the end of the already existing sequences. Therefore, by construction, the new store γ'_{ST} satisfies $\gamma_{\text{ST}} \sqsubseteq \gamma'_{\text{ST}}$.*

The previous proposition shows us that the global store used to save dynamic values increases monotonically during execution. The following results prove that this property strongly simplifies the task of checking the equivalence of configurations.

Proposition 2 *Consider two configurations $\gamma, \gamma' \in \mathcal{C}$ with $\gamma = \langle \gamma_s, \gamma_h, \Gamma \rangle$ and $\gamma' = \langle \gamma'_s, \gamma'_h, \Gamma' \rangle$ such that $\Gamma \sqsubseteq \Gamma'$. Then if $\gamma_s = \gamma'_s$ and $\gamma_h = \gamma'_h$, we have that $\gamma \cong \gamma'$.*

Proof 2 *Let the identity map be the renaming function ρ . By hypothesis, $\gamma_s = \gamma'_s$ and $\gamma_h = \gamma'_h$. Since $\Gamma \sqsubseteq \Gamma'$, by definition, $\forall t \in \mathcal{T}, h, m \in \mathbb{N}$ such that $m \leq len(\Gamma(t,h))$, it holds that $m \leq len(\Gamma'(t,h))$ and $\Gamma(t,h) \downarrow m = \Gamma'(t,h) \downarrow m$, which proves that $\gamma \cong \gamma'$.*

Propositions 1 and 2 allow us to assume that during the analysis of a C program, the store represents a global memory area that is monotonically increased. System states are simply constituted of the *static* and *heap* components of each configuration, and it is correct to only test these two components to know whether two configurations are equivalent.

4 Implementation

In this section we discuss a real implementation of our proposal and we show how we can verify properties in a real example using our tool C.OPEN [7] [5] that lets us to verify C code with the CADP environment.

CADP is a tool box for constructing and analyzing distributed protocols. It uses different formalisms as input, such as LOTOS or BCG (Binary Code Graphs). Moreover, CADP provides OPEN/CÆSAR, a set of libraries to extend the environment with new tools or even to accept new formalisms or languages as input. Figure 2 shows the OPEN/CÆSAR scheme extended to take C programs as input as well as the traditional LOTOS or BCG input. Basically, it divides the functionality of tools into three different modules: the graph module, the storage module and the exploration module. The graph module provides an implicit LTS that gives the necessary functions to deal with the system states, and labels and to evolve from one state to another one executing a transition. Finally, the exploration module uses the storage module to carry out the desired analysis over the system implicitly represented by the graph module. In particular, the tool EVALUATOR is a model checker constructed with this methodology.

Our tool C.OPEN has been designed to analyze concurrent systems developed with C languages and external functions provided by APIs. Given a C application, C.OPEN generates an implicit LTS that can be used as a graph module in OPEN/CÆSAR, so we can use its model checking facilities. For this purpose, the LTS generated is oriented to the analysis of properties when using a well defined API. Now, we consider pointers and dynamic memory as a new feature with its well defined API. Hence, we must consider operations like allocate, deallocate and access to objects as external functions. Following [7], we have extended C.OPEN to implement the operations defined in Section 3. In particular, we have implemented the models for `malloc()` and `free()` external functions, as well as the operation for reading and writing over the dereferencing pointer operator (operator $*$).

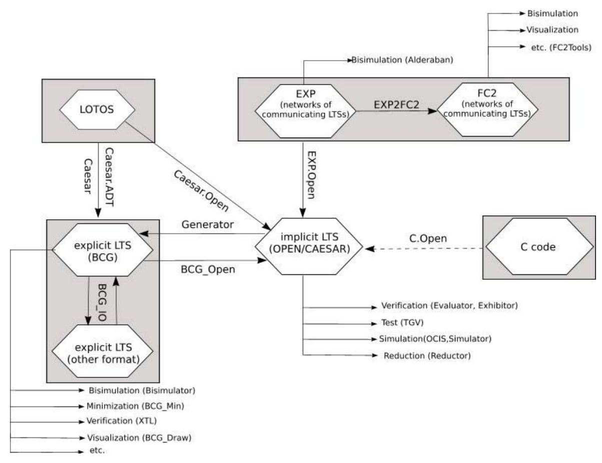

Figure 2. Schema of the extended CADP architecture including C.OPEN

Note that the model for dynamic memory management proposed in this paper can be implemented in any model checker; however, some of them (like SPIN and CADP) are designed to be easily extended by embedding C code. In particular, using CADP we can easily implement some remaining operations on pointers, like referencing (operator &) and pointer arithmetic.

5 Experimental results

In order to evaluate our model and its implementation, in this section we show the results with the verification of the C code to reverse a list. This is a case of study used before in the context of checking properties with model checking dynamic memory (see [14] and [2]). Now, we use it to have performance measurements in the implementation.

```
 struct list{ int data; struct list *next; }; void main(){
    struct list *list;
    struct list *ele;
    struct list *aux;
    int count;
    int next;
    //we create the list
    count=1;
    list=(struct list *)malloc(sizeof(struct list));
    ele=list;
    ele->data=0;
    ele->next=0;
    while(count<100){
        aux=(struct list *)malloc(sizeof(struct list));
        aux->data=cont;
        aux->next=0;
        ele->next=aux;
        ele=aux;
        count=count+1;
    }
    //we get the reversed list
    ele=0;
    while(list!=0){
        aux=ele;
        ele=list;
        list=list->next;
        ele->next=aux;
    }
}
```

Figure 3. Program that reverses a list

elements	states	time	memory	state size
10	93	0:0.226 s.	12M	432B
30	273	0:0.226 s.	15M	752B
50	453	0:0.226 s.	22M	1072B
70	633	0:0.226 s.	28M	1392B
100	903	0:0.601 s.	28M	1872B

Table 1. Verification results for one process

elements	states	time	memory	state size
10	8469	0:0.3 s.	20M	620B
30	74529	0:2.2 s.	120M	1260B
50	205209	0:8.7s.	420M	1900B
70	400689	0:15:1 s.	1100M	2540B
100	815409	1:05 m.	2.8 GB	3500B

Table 2. Verification results for two processes

The C code to reverse the list is shown in (figure 3). The code is divide into two phases: The first one, the creation of the list, will add a new heap entry with every call to `malloc()` function and will add the elements in the store structure every time we assign a value to a dereferencing pointer. The second phase, the reversing of the list, will not add any new elements to the heap structure, but it will modify the heap and the store in order to modify a part of elements of the list during the reversing operation (it modifies the `next` part). Figure 1 in Section 2 shows how the list is internally represented.

Table 1 shows the experimental results of using C.OPEN with EVALUATOR in order to obtain the number of states generated during the verification process, as well as the time taken and the usage of memory. Table 2 contains the results when interleaving two processes doing the same reversing work over two different lists. This second scenario is important to consider backtracking and sharing the global store.

The experiments have been carried out for lists of 10, 30, 70 and 100 elements. In the analysis of a single process, the number of states generated in each configuration is very low; hence the memory needed. Nevertheless, when we experiment with two processes to get a higher number of states due to the interleaving, we obtain high memory usage, specially for more than 50 elements. Note that we only need 15 seconds to analyze 400000 states in the second configuration.

We are currently working on extending the method to deal with very large states. For instance, two processes with lists containing 100 elements produce a state of 3,5KB, excluding the global store. One direction to reduce this problem is to apply the collapse to the heap again, in such a way that only one reference is used to identify the heap. That means that we consider the heap of every process as an element in the global store.

6 Conclusions

We have presented a method to extend model checking tools in order to verify programs with pointers and dynamic memory. The method has been formalized and implemented. We provide a tool, C.OPEN, that can verify realistic concurrent C programs (see `http://www.lcc.uma.es/gisum/tools/smc/` and `http://www.inrialpes.fr/vasy/cadp/software/` for more information about C.OPEN in CADP.)

We have considered a specific framework to evaluate the technique: C programs as input and CADP as the verification platform. However, our proposal can be implemented in different contexts. One future line of work is the extension of SPIN in the same way, extending our previous work presented in [4].

Another line of work planned is the extension of CADP and SPIN property languages to specify and verify properties related to dynamic structures.

References

[1] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, 2001.

[2] A. Bouajjani, P. Habermehl, P. Moro, and T. Vojnar. Verifying Programs with Dynamic 1-Selector-Linked Structures in Regular Model Checking. In *Proc. of 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05)*, 2005.

[3] 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 '00: Proceedings of the 22nd international conference on Software engineering*. ACM Press, 2000.

[4] P. de la Cámara, M. M. Gallardo, P. Merino, and D. Sanán. Model checking software with well-defined APIs: The Socket Case. In *FMICS '05: Proceedings of the 10th international workshop on Formal methods for industrial critical systems*, 2005.

[5] M. del Mar Gallardo, P. Merino, C. Joubert, and D. Sanan. On-the-fly model checking for C programs with extended CADP in FMICS-jETI. In *ICECCS '07: Proceedings of the 12th IEEE International Conference on Engineering Complex Computer Systems (ICECCS 2007)*, 2007.

[6] C. Demartini, R. Iosif, and R. Sisto. dSPIN: A Dynamic Extension of SPIN. In *Proceedings of the 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking*, 1999.

[7] M. Gallardo, P. Merino, and D. Sanan. Towards model checking C code with Open/Caesar. In *Proc. of MSVVEIS'06*, pages 198–201, 2006.

[8] P. Godefroid. Software Model Checking: The VeriSoft Approach. *Form. Methods Syst. Des.*, 26(2), 2005.

[9] K. Havelund and T. Pressburger. Model Checking JAVA Programs using JAVA PathFinder. *STTT*, 2(4), 2000.

[10] G. J. Holzmann. State Compression in SPIN: Recursive Indexing and Compression Training Runs. In *Proc. of the 3th International SPIN Workshop*, 1997.

[11] G. J. Holzmann and M. H. Smith. Software model checking: extracting verification models from source code. *Software Testing, Verification & Reliability*, 11(2).

[12] H. Kastenberg and A. Rensink. Model Checking Dynamic States in GROOVE. In *SPIN*, pages 299–305, 2006.

[13] F. Lerda and W. Visser. Addressing dynamic issues of program model checking. In *SPIN '01: Proceedings of the 8th international SPIN workshop on Model checking of software*, 2001.

[14] A. Møller. Verifying Programs that Manipulate Pointers: (Invited Talk in INFINITY 2003, the 5th International Workshop on Verification of Infinite-States Systems). *Electr. Notes Theor. Comput. Sci.*, 98, 2004.

[15] A. Møller and M. I. Schwartzbach. The pointer assertion logic engine. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, 2001.

[16] M. Musuvathi, D. Y. W. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: a pragmatic approach to model checking real code. *SIGOPS Oper. Syst. Rev.*, 36(SI), 2002.

[17] J. C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS*, 2002.

[18] Robby, M. B. Dwyer, and J. Hatcliff. Bogor: an extensible and highly-modular software model checking framework. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, 2003.