

Extracting Zing Models from C Source Code^{*}

Tomas Matousek and Filip Zavoral

Charles University in Prague
Department of Software Engineering
Malostranske namesti 25,
11800 Prague, Czech Republic
{Tomas.Matousek, Filip.Zavoral}@mff.cuni.cz

Abstract. In the paper, we propose an approach to an automatic extraction of verification models for the C language source code. We primarily focus on the representation of pointers and arrays, which make the extraction from the C language specific. We provide an implementation of the model extractor as a part of our broader effort to develop a verifier of Windows kernel drivers based on the Zing model checker. To demonstrate the feasibility of our approach, we give examples of the extraction results on a practical synchronization problem.

1 Introduction

A concurrent program working in a critical environment, such as a Windows kernel driver [14], can contain various implementation errors that may lead to a failure of the entire system. Errors range from simple ones that can be easily found by means of software testing up to errors that are very difficult to discover as they appear only at certain irreproducible circumstances, e. g. when threads are scheduled in a particular order. Impossibility to emulate the exact state of the complex environment makes it even more difficult to check whether an error is present.

Many existing verification tools based on static analyses and/or testing at run-time, including *PREfast* [12], *CCured* [19], *ElectricFence* [20] etc., target errors that stem from incorrect use of the programming language constructs (e. g. accessing unallocated memory, dereferencing a null pointer, a buffer overrun, and so on). Even though no such errors are present in the program, the program need not to interact well with the surrounding environment. It is therefore necessary to verify that the program complies with the rules the environment imposes on the use of the provided functionality. An example of such requirement is the restriction on the order of calls to environment functions. Using the environment specification and the source code of the program, tools like *Microsoft Static Driver Verifier* [13], *Bandera* [3] and *Java Path Finder* [17] generate a model

^{*} The work was partly supported by the project 1ET100300419 of the Program Information Society of the Thematic Program II of the National Research Program of the Czech Republic.

of the interaction between the program and the environment and verify that given properties hold on every possible execution path via a technique of *model checking* [2].

The model checking is a formal verification method based on thorough exploration of the model emulating the software unit behavior with respect to a given property. The model should be as simple as possible since the model checker explores all the possible states of the model. The time and space requirements for the verification are growing exponentially with respect to the number of operations, threads and variables of the model (the *state explosion problem* [10]).

The target modeling language for our model extractor is the *Zing language* [1, 16], being developed by Microsoft Research group on the top of the *Microsoft .NET Framework platform* [11]. The choice was made due to the rich modeling functionality the Zing language provides and the state of its current development – the preview implementation of the model checker is available and works quite well. However, most ideas behind our work are independent of the particular target model checker and can be applied to any modeling language that provides for unbounded heap allocation, unbounded call-stack and dynamic thread creation. Another modeling language meeting these criteria should be the *Bandera Intermediate Representation* (BIR) – a modeling language of the *Bogor* model checking framework [21].

1.1 Paper Contribution

We propose a novel approach to the extraction of models from C source code and provide the implementation targeting the Zing model checker. Existing works either focus on Java-like languages (e. g. Bandera [3], Java Path Finder [17]), do not extract the model fully automatically (e. g. *SPIN* [6]) and/or are very limited on the constructs that can be used in the source code (e. g., *SPIN* does not support unbounded heap allocation, call stacks nor dynamic thread creation).

The rest of the paper is structured as follows. Section 2 presents the key ideas on the model extraction and explains the approach we take for modeling various C language constructs. Section 3 discusses the related work and Section 4 concludes and outlines the future work.

2 Modeling C Programs in the Zing Language

The ultimate goal of our broader work is to develop a model extractor of Windows kernel drivers that produces Zing models, which could be passed to the Zing model checker for verification. The drivers are usually written in the C language with Microsoft extensions. Therefore, we expect the source code to comply with the C language specification [7] extended by features provided by the Microsoft C compiler including, for example, the structured exception handling.

Many C language features do not map to the Zing straightforwardly. These include pointer and array operations, unions, casts, as well as the way in which the source code is compiled and linked. We consider pointers and arrays the

main issues and hence focus on how to model them by the means of the Zing language in this paper.

2.1 Modeling Types

The model extractor distinguishes data types available in the C language among primitive types (void, integers, floats and string literals), composite types (structures and unions), static arrays, and data and function pointers.

Except for string literals, primitive types map to the Zing language one to one. String type is not available in the Zing language so the literals are represented as arrays of integers.

C structures can easily be mapped to the Zing language as it supports structures as compound value types¹. However, C structures can behave not only as value types, when allocated statically, but also as reference types, when allocated dynamically on the heap. Therefore, the structures need to be enclosed into the classes in the latter case. The process of enclosing is similar to the boxing known from C# or Java – we also refer to the enclosing classes as to *boxes*. Structures are not the only types that may require boxing. In general, any *value type* (i. e. integers, floats, composite types and pointers) may require boxing under certain circumstances explained later.

The Zing class that implements the box is denoted *Box<T>* in the further text, where *T* is the Zing type being boxed. The class contains a single field of type *T* named *Value* that holds the boxed value. Since the Zing language does not support parametric polymorphism, the model extractor emits Zing code for all different constructed types used throughout the model.

2.2 Modeling Variables

Before the model is generated, the model extractor performs a simple analysis to determine for each variable (i. e. for each field, local variable, global variable and formal parameter) whether its address is ever taken. To get the address-may-be-taken information, it is sufficient to maintain initially cleared flag for each variable and set it whenever the *address-of* operator is applied to the variable.

The model extractor distinguishes three *variable models* depending on the type of the variable and the results of the previous analysis:

1. The variable model is *Value* if the variable is of an integer, float, compound type or any pointer type and its address is never taken.
2. The variable model is *BoxedValue* if the variable is of an integer, float, or any pointer type and its address may be taken.

¹ There are some known implementation issues related to the structures in the latest version of Zing. The extractor embeds fields of structures into their containers in order to produce models that can be checked in the current model checker. For clarity, let us assume the structures work as expected in this paper.

3. The variable model is *StaticArray* if it is of a static array type with constant length (whether or not its address is taken). The Microsoft C compiler does not support static array types of a variable length except when used as a flexible array member. The model extractor does not support this feature yet.

The variable model determines how the variable is represented in the model and how operations applied on the variable are modeled.

1. Non-pointer variables of the *Value* model are converted to Zing variables of types corresponding to their C types. Operations on them are modeled by the corresponding Zing operations.

Variables of data pointer types are modeled by Zing variables of type *Pointer*, which is a Zing structure introduced by the model extractor to represent a data pointer of any type. Operations on these variables (dereferences and pointer arithmetic) are translated to calls to the auxiliary atomic methods *DerefGet<T>*, *DerefSet<T>*, *AddIntPtr*, *SubPtrPtr* and *CmpPtrPtr*.

Variables of a function pointer type are converted to Zing variables of an integer type. The values are integer constants identifying the functions that can be called via the pointer. The indirect call operation is modeled by a switch statement. The number of cases of the switch statement may be reduced by utilizing results of a *points-to analysis* [5], which can determine the superset of possible targets of each pointer (in the worst case, any function whose pointer is ever taken and whose signature is compatible with types of actual arguments).

2. Variables of the *BoxedValue* model are declared as Zing variables of the *Box<T>* type, where *T* is a value type. They are initialized by a new instance of the *Box<T>* class. Operations on these variables are the same operations as on variables of the *Variable* model yet enriched by the *Value* field access.
3. Variables of the *StaticArray* model are declared as Zing variables of an array type, which we denote *array<T>*. Multi-dimensional C arrays are flattened to vectors represented by Zing arrays. Arrays of arrays are never used. The flattening is necessary since the C language allows casting among arrays of different shape. Such operation is empty on flattened arrays.

The flattening also enables a C pointer to a static array of a fixed size to point anywhere into the multi-dimensional array. The operations on static arrays (reading or writing an element addressed by an index) are converted to Zing array operations and a calculation of the index if the C array is multi-dimensional.

2.3 Modeling Data Pointers

In the C language, a data pointer can point to statically or dynamically allocated memory. The static allocation reserves the memory implicitly on the stack

or within dynamically allocated memory. Dynamically allocated memory is a storage on the heap allocated explicitly by a call to an *allocator* routine of the environment. The environment may provide various allocators differing in the kind of memory they allocate (e. g., several different memory pools are available for drivers in the kernel mode). The model extractor expects the specification of the environment, written in the *DeSpec language* [9], to mark allocator functions explicitly.

Each allocator should return an instance of the class *Memory*, which represents allocated uninitialized raw memory in the generated model. The only field of this class significant for pointer modeling is the *Object* field of the *object* type. The field is initialized to a *null* reference when returned from the allocator, expressing the fact that the memory is uninitialized, and filled in later when the memory is written to by the C program. As the dynamically allocated memory is accessible only via pointers, it is the responsibility of the pointer operators to fill the field appropriately.

Although there are substantial differences between the dynamically and statically allocated memory, pointers in the C language do not make difference. Hence, all pointers should be assignable one to another regardless of their target. In the model, we represent the pointer of arbitrary data pointer type as a pair (structure) $\{Offset : int, Target : object\}$, where the *Offset* is an integer greater or equal to -1 and the *Target* is an arbitrary Zing reference².

The extractor distinguishes four kinds of non-null pointers derived from the kinds of storage they point to:

1. A pointer to a statically allocated storage
 - (a) containing a single value (*static single-value pointer*)
 - (b) containing a sequence of contiguous values (*static multi-value pointer*)
2. A pointer to a dynamically allocated storage
 - (a) provably containing a single value (*dynamic single-value pointer*)
 - (b) possibly containing a sequence of contiguous values (*dynamic multi-value pointer*)

The *null* pointer is represented by the pair $\{Offset = 0, Target = null\}$, a function pointer casted to the void data pointer is encoded as a pair $\{Offset = function_id, Target = null\}$.

The static single-value pointer points to a boxed value (see Fig. 1). The *Target* holds a reference of the type $Box<T>$. The *Offset* is set to the special value -1 , which identifies this kind of pointer at run-time.

The static multi-value pointer points to or into a static array, which is represented by an instance of $array<T>$ as explained previously (see also Fig. 2). The *Target* always holds a reference to the array and the *Offset* contains a non-negative integer value designating the item of the array to which the pointer actually points.

² The *Caduceus* tool [8] takes a similar approach to the pointer representation for reasoning about aliased variables by theorem proving.

```
int t = 1;
int *s = &t;
```



Fig. 1. Use of a static single-value pointer in the C source code and the corresponding Zing model.

```
int a[5];
int (*u)[2] = &a[1];
int *v = a;
u[1][1] = 3;
v += 4;
*v = 6;
```

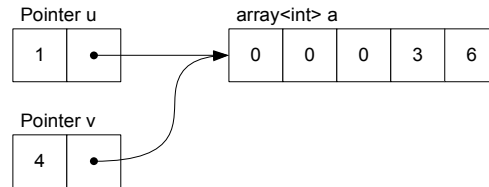


Fig. 2. Use of static multi-value pointers and the corresponding Zing model in the final state.

Dynamic pointers are always initialized by an assignment from the return value of an allocator. At that point, the model extractor may not know the particular structuring of the raw memory, as the type of the variable where the pointer returned from the allocator is stored to may be the void pointer. Although the model extractor performs a static analysis for discovering the interpretation of the memory, the analysis can reach the decision that the storage is interpreted differently on two possible execution paths. Therefore, the interpretation is deferred to the point where it really happens in the program – to the first write operation to the storage.

We assume the program does not reinterpret the storage once it triggered the write operation with a particular interpretation, for example, by casting one structure to another. If the compiler specific directives explicitly control the layout of the storage the reinterpretation need not to be incorrect. However, the model extractor does not currently support this feature as it is used sporadically. To allow an arbitrary reinterpretation, the *Memory* object would carry not only the value but also its type id. Using this additional information, the dereferencing operations would perform appropriate bit conversions when they reinterpret the content of the storage. The reinterpretation of the statically allocated storage would be implemented similarly.

Since the C language does not explicitly distinguish a pointer to a single value from the pointer to multiple values, the model extractor supposes there may be more values following the value pointed to by the dynamic pointer. If the static analysis is able to determine the interpretation of the storage and the number of bytes the allocator allocates for the storage it is possible to decide whether the storage hosts a single value or some fixed number of multiple values³. If pointers pointing to the storage are provably not involved in any pointer arithmetic or

³ It is also necessary to determine whether some kind of reallocation may be applied on the storage if the environment provides such functionality.

indexing operations, the model extractor can also assume there is only a single value stored in the memory. If neither analysis reaches the conclusion, the model extractor assumes the storage holds an unbounded number of values.

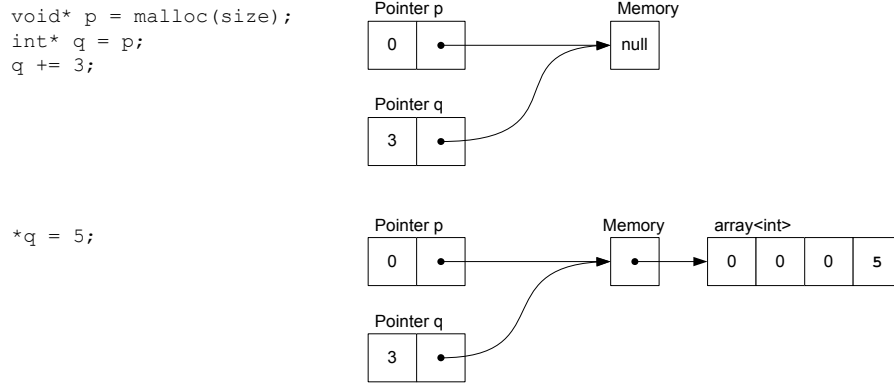


Fig. 3. Use of dynamic multi-value pointers and the corresponding Zing model states during the execution.

The dynamic pointer is initialized so that the *Target* field points to the *Memory* object returned by the allocator (see Fig. 3). The *Offset* is set to the special value of -1 in the case of a single-value pointer and to 0 otherwise, which means the pointer is pointing to the first value stored in the memory. The pointer arithmetic, allowed only on multi-value pointers, change only the *Offset* and never the *Target* field – the *Target* always references the *Memory* object.

The write operation through the dynamic pointer referencing type T firstly checks whether the *Memory* is raw (i. e. the *Object* field is a *null* reference). If so, it initializes the *Object* field to an instance of the `Box<T>` class, in the case of single-value pointer, or to an instance of `array<T>`, otherwise. The value being written by the operation is stored to the *Value* field and to the appropriate slot in the array, respectively.

If the array is not large enough to contain the slot, it is expanded, so that the slot becomes available. The array does not expand exponentially; instead, it expands only enough to provide the slot for the item written by the operation. As the expansion is performed within the atomic write operation, it is better to do more expansions than to enlarge the state space unnecessarily. Since the purpose of the model is not to check whether the program reads from outside the allocated memory, the array is expanded without limits. Checking bounds could be optionally enabled, but would require the model to be enriched by additional information (the actual size of the allocated memory stored on the *Memory* object) making the state space larger.

The read operation through the dynamic pointer returns either the *Value* field of the `Box<T>` class or the content of the appropriate slot from the array.

The array expansion does not take place here, as it is an error to read from an uninitialized storage. Instead, a failing assertion is inserted to the model.

Although pointers of all kinds must generally be represented by the same Zing structure (pointers may be assigned to each other), they are not intermixed in many cases. Passing a value to a function by reference is an example, where static single-value pointers are usually used exclusively. Since the model extractor knows the entire program, it is able to determine whether a pointer variable definitely holds a single kind of pointer throughout the entire execution. If that is the case, the resulting model can be simplified by removing the redundant information provided by the special values of the *Offset* field in the case of single-value pointers.

2.4 Model Size

We made various tests with the extractor including verification of a C implementation of a synchronized priority queue via a singly linked list. In the Windows kernel, some drivers use this kind of queue for queuing and sorting I/O Request Packets (IRPs) for sequential processing. The tested program initializes the queue and starts several threads inserting items to the queue. Finally, when all threads terminate, the program checks the integrity of the queue – whether it is sorted and whether no item got lost by invalid handling of pointers during manipulation with the queue. The C source code has around 110 lines and the entire generated Zing model about 900 lines. All tests were performed on 1.4GHz/1GB machine.

With 2 producers, each inserting 3 items to the queue synchronizing updates only (i. e. the search for the place, where to insert an item, was unprotected), the Zing model checker found the error within 3 seconds – the queue was not sorted correctly at the end of the program.

The next naive implementation used read-write lock so that the queue search routine acquired the lock for write-exclusivity and the subsequent queue update operation upgraded the lock to read-write-exclusivity. After the insertion, the lock was released. The model checker found a deadlock for 5 producers each inserting 3 items in 6 seconds.

The correct version of the program respectively running 2, 2, 2 and 3 producers each inserting 2, 3, 6 and 3 items to the queue passed the verification in 6 seconds, 17 seconds, 2 minutes and 31 minutes. These times suggest that the number of threads has much greater impact than the number of items inserted to the queue, which is positive as the race conditions are usually revealed even for a small number of threads.

Outcomes of the tests we performed, part of which presented above, confirm that the generated models are feasible for verification.

3 Related Work

Various approaches and tools to verify C programs were developed during many years of research.

The Microsoft PREfast tool [12] performs a static analysis of the source code and searches for common error patterns. Its specialized version is also able to discover errors specific to the Windows kernel drivers. The tool can, for example, find memory leaks incurred by missing function calls, dereferences of null pointers, buffer overruns, kernel functions called on an incorrect IRQL level, and so on. The analysis is function scoped and hence introduces false negatives and also restricts variety of errors the tool is able to detect.

The CCured tool [19] instruments the C program with information that allows the run-time checks to discover invalid use of pointers, invalid casts, buffer overruns, and so on. It is built on the top of the *CIL infrastructure* [18] – the system our model extractor utilizes as the front-end. The CCured tool also distinguishes various pointer types to minimize the necessary run-time checks. In contrast to the model extractor, this tool trades the space for the time while the model extractor does the opposite as the increase in operations complexity can be neutralized by the use of the atomic execution.

Neither the PREfast nor the CCured tool allows to verify the correctness of the program interaction with its environment. The *SLAM project* [15] is addressing the static analysis and verification of the C programs, especially the Windows kernel drivers. The beta version of the Microsoft Static Driver Verifier (SDV) tool [13, 15] has been recently released and used in practice to find errors in Microsoft’s own drivers. It enables to check drivers against many rules imposed by the kernel. However, the environment model used by the SDV tool is single-threaded, preventing verification of some race conditions, and quite non-deterministic, introducing additional false reports.

In contrast, using our model extractor, a program written in the C language can be checked against any property that can be encoded into assertions. The model extractor does not limit the environment model in any way. It can be simple or complex, single-threaded or multi-threaded, less or more deterministic, depending on the particular property being verified. The only limitation is the resulting model size.

4 Conclusion & Future Work

We propose an approach to the extraction of verification models from C source code that enable the model checking technique to discover errors contained in C programs. We have implemented a tool, the model extractor, that automatically generates a Zing model from the source code of the program. Finally, we show that the extracted model is feasible for verification in practice.

The major issues of the C program model extraction addressed by this paper stem from pointer and array operations. In our work, we distinguish four kinds of data pointers depending on the kind of memory and the possible number of items they are pointing to. Although this differentiation leads to more complicated dereferencing operations, it minimizes the state space of the model. Due to the atomicity of the dereferencing operations, the complexity increase does not influence the resulting model size. Further improvements of the analysis per-

formed by the model extractor will make possible to go further and discover the cases when the various kinds of pointers need not to be represented uniformly saving more bits of state space.

References

1. Andrews, T., Qadeer, S., Rajamani, S. K., Rehof, J., Xie, Y: Zing: A model checker for concurrent software, Technical report, Microsoft Research, 2004.
2. Clarke, E. M., Grumberg, O., Peled, D. A.: Model Checking, MIT Press, 2000.
3. Corbett, J. C., Dwyer, M. B., Hatcliff, J., Laubach, S., Pasareanu, C. S., Robby, Zheng, H.: Bandera: Extracting Finite-state Models from Java Source Code, proceedings of the International Conference on Software Engineering (ICSE), 2000
4. Hind, M.: Pointer analysis: Haven't we solved this problem yet? In 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01), 2001.
5. Holzmann, G. J.: The SPIN Model Checker: Primer and Reference Manual, Addison-Wesley Professional, 2003
6. ISO: C99 – The C Programming Language Standard (ISO/IEC 9899:1999)
<http://www.iso.org>
7. J.-C. Filliâtre and C. March. Multi-prover verification of C programs. In J. Davies, W. Schulte, and M. Barnett, editors, Sixth International Conference on Formal Engineering Methods, volume 3308 of Lecture Notes in Computer Science, pages 15-29, Seattle, WA, USA, Nov. 2004. Springer-Verlag.
8. Matousek, T.: Model of the Windows Driver Environment, Master Thesis at Department of Software Engineering, Charles University in Prague, 2005,
<http://nenya.ms.mff.cuni.cz/publications/Matousek-thesis.pdf>
9. McMillan, K. L.: Symbolic model checking – an approach to the state explosion problem, PhD thesis, SCS, Carnegie Mellon University, 1992
10. Microsoft: .NET Framework, MSDN,
<http://msdn.microsoft.com/netframework>
11. Microsoft: PRefast,
<http://www.microsoft.com/whdc/devtools/tools/PRefast.msp>
12. Microsoft: Static Driver Verifier – Finding Driver Bugs at Compile-Time, WHDC,
<http://www.microsoft.com/whdc/devtools/tools/sdv.msp>
13. Microsoft: Windows Driver Foundation, WHDC,
<http://www.microsoft.com/whdc/driver/wdf/default.msp>
14. Microsoft Research: SLAM Project, <http://research.microsoft.com/slam>
15. Microsoft Research: Zing Model Checker, <http://research.microsoft.com/zing>
16. NASA Intelligent Systems Division: Java Path Finder,
<http://ase.arc.nasa.gov/havelund/jpf.html>
17. Necula, G. C., McPeak, S., Rahul, S. P., Weimer, W.: CIL: Intermediate Language for Analysis and Transformation of C Programs, Proceedings of Conference on Compiler Construction, 2002.
18. Necula, G. C., McPeak, S., Weimer, W., Harren, M., Condit, J.: CCured,
<http://manju.cs.berkeley.edu/ccured>
19. Perens, B.: ElectricFence, <http://perens.com/FreeSoftware/ElectricFence>
20. Robby, Dwyer, M. B., Hatcliff, J.: Bogor: An Extensible and Highly Modular Software Model Checking Framework, SIGSOFT Softw. Eng. Notes 28, 5, 267-276, 2003