The Development of Automatically Verifiable Systems using Data Representation Synthesis

Bryce Cronkite-Ratcliff
Department of Computer Science, Stanford University
brycecr@stanford.edu

ABSTRACT

We present a technique for building systems that use arbitrary data relationships and yet are more easily verified than systems that use pointer data structures. In particular, by replacing pointer data structures with structures generated by Data Representation Synthesis (DRS), we are able to use automatic techniques for verifying container-manipulating code to verify the safety of a complete system. We rebuild an existing microkernel with DRS-generated structures and demonstrate a reduction in data-structure manipulating lines of code, good performance, and automatic verifiability of safety properties.

1. INTRODUCTION

In order to enable the construction of robust software systems in all developent contexts, there is a pressing need for techniques to develop provably correct programs. One of the greatest challenges in formal verification lies in automatically reasoning about pointer data structures, where issues of aliasing – that multiple symbolic names may refer to the same memory location – and indirection greatly complicate automated reasoning [11].

While this challenge still stands, progress has been made on the related problem of analyzing containers, a class of data structures that permit insertion, query, removal, and iteration operations. In particular, work by Dillig et al. has provided a general theory for automatic reasoning about containers, and the authors have implemented this work into a functional cross-platform automatic verifier called Compass [1]. Thus, if we can change the way software is written such that containers are used instead of pointer data structures, we can take advantage of Compass' precise reasoning about containers to automatically verify the resulting code.

One approach to building code without pointer data structures is Data Representation Synthesis (DRS). In DRS, the programmer provides a high-level description of the data as mathematical relations where this specification does not commit to any particular implementation. Then, a compiler selects a particular data structure implementation for the specified relation. As a result, the correctness of data structure operations is now by construction. Most relevant to this work, however, is that DRS encapsulates arbitrary data handling in an interface that supports operations that can be expressed in terms of operations on containers. Thus, we can use the precise reasoning techniques for containers already mentioned to perform automatic verification of systems built on structures generated by Data Representation Synthesis.

Thus, the use of Data Representation Synthesis and techniques for precise automatic reasoning about containers can provide automatically verifiable complete systems that are performant and simple to build and maintain. The goal of the work presented in this paper is to demonstrate these claims by using DRS to build a complex, performance and safety critical system, and demonstrate feasibility, good performance, and automatic verifiability.

2. BACKGROUND

2.1 Precise Reasoning about Containers

Containers are a class of abstract data structures that allow elements to be inserted, retrieved, removed, or iterated over. A wide range of familiar data structures – including maps, vectors, lists, sets, stacks, and deques – are containers, including the heavily used structures provided by the Java Collections Framework and C++ Standard Template Library. The widespread use of standard container interfaces suggests, to the student of program analysis, two things: first, that automatic analysis of code using data structures is valuable, as it greatly increases the scope of programs that may be automatically analyzed, and, second, that automatic analysis of container client programs can be decoupled from the potentially more tedious task of verifying the the particular container implementation; one implementation analysis can serve for all the clients of the container [1].

Recent work has developed a technique for precise automatic static analysis of container-manipulating programs, which we will refer to as Precise Container Reasoning (PCR) [1]. PCR differs from previous approaches to container analysis in that it separates containers into position-dependent containers (such as stacks, vectors, lists) and value-dependent containers (such as maps and sets), and provides a constraint-based means to reason about key-value and position-value relationships for value-dependent and position-dependent containers, respectively. In tests conducted by the authors, PCR found all container usage errors detected by a technique that did not consider these key-value and position-value relationships and produced many fewer false positives, to the point where false positives never outnumbered actual errors and usually accounted for far fewer warnings. Compass is an analysis application that implements PCR to allow for automatic analysis of container-manipulating programs [1]. Compass currently supports C++.

2.2 Relations as Containers

To a first approximation, a relation can be seen as a container of the tuples in the relation, where queries on the attributes serve as the indices into the relation. Thus, if we can express relational operations as operations on containers, we can use the automatic approach to verifying code that manipulates containers implemented in Compass.

More formally, we are interested in demonstrating the claim that any operation on a relation can be represented as a set of operations on a set of containers.

2.2.1 Operational Primitives

We first define a set of symbols for expressing relational operations; this is drawn from the description for these symbols found in [3]. A set of values v is drawn from a universe of values V that includes the integers. A tuple $t = (a_1 : v_1, a_2 : v_2, \ldots)$ maps a set of attributes (also called fields or columns) $\{a_1, a_2, \ldots\}$ to values from V. We use $t(a_i)$ to denote the particular value of attribute a_i in tuple t. If $t(a_i) = s(a_i)$ for all attributes in s, we say t extends s, expressed as $t \subseteq s$. Similarly, if $t(a_i) = s(a_i)$ for all common attributes, s matches t, expressed as $s \sim t$. $s \lhd t$ gives the merge of tuples s and t, where we take the value of $s(a_i)$ if $t(a_i) \neq s(a_i)$. A relation r is a set of tuples over the same set of attributes S_A, also called the attribute schema. Note that all relational algebraic operations have their usual meanings [10].

Let us first enumerate the relational operations that may be provided by structures generated via Data Representation Synthesis, here provided in terms of precise relational algebraic operations just described.

empty () = ref $\emptyset$
insert r t = r $\leftarrow$!r $\cup$ {t}
remove r s = r $\leftarrow$!r $\setminus$ t $\in$!r | t $\subseteq$ s
update r s u = r $\leftarrow$ {if $\subseteq$ s then t $\lhd$ u else t | t in !r}
query r s C = π_C\{t $\in$!r | r $\subseteq$ s\}

Informally, these operations can be described as follows:

- empty () returns a new empty relation
- insert r t inserts a tuple t into relation r
- remove r s removes a tuple s from relation r
- update r s u updates a tuple t from relation r that matches tuple s with the attributes in tuple u
- query r s A returns a new relation with the attributes A of all tuples in r that match tuple s

In the above, tuples s and u may be partial tuples; tuple t is a completely specified tuple.

Again, our objective in this work is to express how each of the relational operations above can be expressed in terms of operations on containers, enumerated below after the model provided by Dillig, Dillig, & Aiken [1]:

- c.**read**(k) // read the value associated with key k from container c, or null if no value is associated with k
- c.**write**(k, v) // insert v as the value for key k in container c
- **foreach**(k,v) in c do e od // for each key value pair in container c (i.e. v is non-null), execute expression or set of expressions e

Here k, v, c, e represent a key, value, container, and expression, respectively, and c.g() means that the function g is called on container c.

In short, containers are modeled simply as key-value stores where elements can be written, retrieved, or iterated over. Note that we reserve some special identifier null to indicate that a value with a particular key is not present in the container.

2.2.2 Relations as Containers: General Case

First, in the following description, all containers are taken to be *value-dependent*; they should be thought of as simply key-value stores where elements are inserted as key-value pairs, and retrieved or updated by keys.

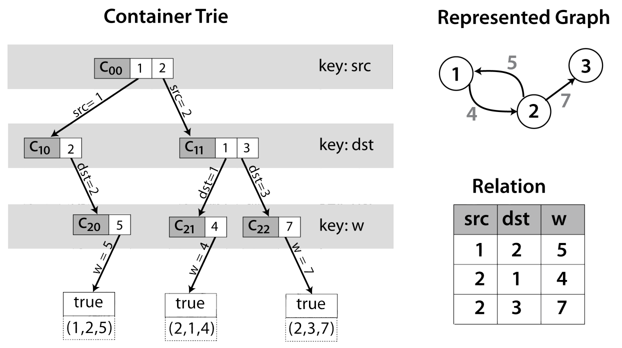
Consider a relation R and a nested set of value-dependent containers C. R and C contain exactly the same data, albeit organized in a different fashion.

C can be thought of as a trie where each edge represents a particular value of an attribute of R. All of the edges between a level l and $l{+}1$ of the trie represent different values for the same attribute a_l in R. The number of levels in the trie is $|S_R| + 1$ where $|S_R|$ is the cardinality of the attribute schema of R. Each non-leaf node n_{li}, where l is the depth of the node in the trie, is implemented as a container c_{li} where keys are particular values of an attribute of a_l; indexing into this container with a particular value v_l returns either the node $n_{(l+1)j}$ connected to n_l by an edge with value v_l or null if no such edge exists. Leaf nodes represent full tuples in the relation. Notably, the containers represented by nodes at the level $l = |S_R| - 1$ (the parents of the leaves) need only map keys to some non-null value, which we take to be true). Note also that this model implies a nesting order for the containers in C – to index into a container with key v_{l+1}, one first must index into a container with key v_l. Please see Figure 1 in the next section for an illustration of an example container trie.

2.2.3 Relations as Containers: An Example

For an illustrative example, consider a relation R representing a directed graph G and corresponding set of nested containers (a container trie) C. Each tuple represents an edge and has the relation is on three attributes: a source node src, a destination node dst, and a weight w.

Arbitrarily, we choose the following order (and thus container nesting in C) for the attributes of R: $a_0 = src$, $a_1 = dst$, and $a_2 = w$. C consists of a set of containers c_{li} where containers of c_{li} take as keys entries of attribute a_l and i is simply an index to differentiate containers of the



src	dst	w
1	2	5
2	1	4
2	3	7

Figure 1: The three diagrams above give three ways of representing the same information: as weighted, directed graph, as a relation (simply a table) and as a relation stored in a container trie. Note that the tuple values in in the leaves of the trie are not explicitly stored, but are implicit in the path taken to reach the particular leaf.

same depth. When a container trie is not empty there is always a container c_{00} that is the "entry point" to C – it is always the first container indexed.

The relation starts empty. Consider that we wish to insert an edge $e = (1, 2, 5) = (src^*, dst^*, w^*)$ into the relation.

First, we note the container is empty, and thus create a new container to act as the first container in C, c_{00}. We then read the value for key 1 from c_{00}. The returned value is null; this is the first edge of source src^* in R, so we create a new container c_{10} and insert the key-value pair $< 1, c_{10} >$ into the container c_{00}. We read the value for key 2 from container c_{10} and again find null, so we create a new container c_{20} and insert the key-value pair $< 2, c_{20} >$ into the container c_{10}. Now, we read the value for key 5 from container c_{20} and again find null, and we insert the key-value pair $< w^*, \mathsf{true} >$ into the container c_{20}. Edge e is now present in R.

Now consider that we have inserted several more edges into R, such that it looks like Figure 1 and wish to service a request such as retrieving the src for all edges with $w = 5$ – that is, the operation query R s=(*,*,5) A=$\{src\}$. Our tuple to match, s, does not specify a value for src or dst fields, so we must issue a foreach loop over all the elements in the root container. Thus, we follow each edge from the root container to a child; we first look at c_{10} Again, s does not specify a value for dst, so we loop over the key-value pairs and travel the edge with $dst = 2$ from c_{10} to c_{20}. We note c_{20} contains a pair $(5, true)$, which matches our tuple s, so we record the tuple $(1, 2, 5)$, as those were the keys that brought us to this matching $true$ leaf. Then, we repeat the same procedure for $src = 2$, but we find no leaves for which the key 5 at level $l = 2$ (the level where container keys are values of w) is $true$. We now project the only matching tuple we found $(1, 2, 5)$ on to the attributes in A, which is just src. That is, we create a new tuple $t = (1)$ and insert this tuple into a new relation backed by a container trie, D. We return D to the programmer and are finished.

2.3 Data Representation Synthesis

Data Representation Synthesis (DRS) is a programming tool designed to provide a means of decoupling the data structure interface from the data structure implementation [3]. In DRS, the programmer provides a *relational specification* and *decomposition*. The relational specification is a description of the data to represent, how each piece of data relates to others (for example, which attributes act as relational keys), and which relational operations should be supported. The decomposition provides a description of how the data structure should be implemented as a combination of existing data structure primitives. Thus, the programmer defines the interface to their data structure via a relational specification, and this definition is entirely separate from the definition of the backing decomposition.

DRS has been implemented by Hawkins in the form of RelC. RelC takes as input relational specifications and decompositions written in a simple OCAML-like syntax and generates data structures in C++ or Java that implement the requested relational specification using the supplied decomposition.

Finally, because DRS generates high-performance encapsulated data structures that can represent arbitrary relations, hand-coded pointer data structures are no longer necessary, and we can use existing techniques for precise reasoning about containers to reason about DRS-generated relations. Thus, we have a promising approach to the problem of automatic analysis of programs using any data relationships.

3. VERIFYING THE SAFETY OF A MICROKERNEL

3.1 Base System: the Fiasco Microkernel

3.1.1 Choosing a System

A proof of concept system for the use of Data Representation Synthesis should: show that RelC-generated structures can be used in complex applications, and that their use in fact simplifies the code base; demonstrate that the ported system operates with minimal overhead; and demonstrate that the ported system is easier to formally verify than the original system (using PCR). We searched for a system for which demonstrating each of these goals had concrete significance.

An operating system kernel is a highly complex system where performance and safety are absolutely critical. A kernel thus easily fulfills the three qualifications required for a base system. The Fiasco.OC microkernel (referred to in this paper as Fiasco for simplicity), is a microkernel developed at the Dresden University of Technology. Fiasco has the particular advantages that its source code is openly available, it is written in C++ (which the RelC backend is capable of producing), and it is reasonably small (and thus a port is within scope for this research project) with some 15,000 lines of code [9].

3.2 Porting the kernel

In order to automatically verify the kernel, it was necessary to replace all pointer data structures – which Compass cannot reason about – with relations generated by RelC. The pointer data structures in Fiasco include open-coded lists, treemaps, priority queues, and arrays, among others. The process for porting each of these structures was as follows:

1. Determine a relational specification for the structure. Most structures in Fiasco can be modeled as simple key-value stores.
2. Determine an appropriate decomposition for the structure, considering ordering, access patterns, and overlap of structure primitives between kernal data structures.
3. Generate a C++ structure with RelC using as input the relational specification and decomposition developed.
4. Convert this generated data structure into the Fiasco dialect of C++.
5. Splice the new structure into place and remove the old pointer data structure.
6. Verify correct behavior by running tests on Fiasco and examining kernel state during operation.

Working within the contained environment of the kernel source code imposed a number of unique constraints on the above procedure, some of which are described below:

Becuase the C++ generated by RelC differed from the dialect of C++ in which Fiasco is written an extra (manual) translation step was necessary.

Further, the data structure primitives supported by RelC out-of-the-box were those implemented in the STL and Boost libraries. Both of these frameworks are difficult to splice into minimal required subsets, and both make extensive and complex use of C++ language features disallowed in Fiasco. Thus, new templatized data structure primitives independent of any libraries were developed. In addition, the RelC-to-data structure primitive layer was rewritten to mimize Boost dependence.

RelC-generated relations use `malloc` to request memory for new objects; memory requests had to be redirected to go through the kernel's memory allocator instead. This engendered a circular dependency when porting the structures that backed the kernel memory allocator. The solution was to statically allocate the memory required to support these relations, including using data structure primitives implemented with statically allocated arrays.

Once a port of the kernel was achieved, correct operation was tested by executing benchmarks and running a paravirtualized linux, L4Linux, on the ported kernel. Unfortunately, more elaborate tests of correct kernel behavior are not currently available for Fiasco, but the techniques used to ensure correct performance were the same used by the developers of the kernel.

3.3 Performance Evaluation

While the primary thrust of this work is to demonstrate verifiability, we acknowledge the importance of performance, especially in low-level systems like microkernels. We ran simple benchmarks stressing different ported modules of the kernel system. Each benchmark was written in C and run in userspace on top of the ported kernel. Each benchmark was repeated several times to account for fluctuations in runtime.

Benchmark	O(avg msec)	R(avg msec)	O σ	R σ
Pthreads	346	366	232	191
Memory	4689	4868	36	18
IPC	45146	48036	1050	1128

Table 1: Performance results from kernel benchmarks. O is original kernel; R is relational port.

The Pthreads benchmark consisted in creating a set of pthreads, assigning each with a simple task, and then joining all dispatched pthreads. The Memory benchmark consisted of allocating, touching, and freeing blocks of memory. The IPC benchmark consisted of passing strings back and forth among a few separate threads using the kernel's provided Inter-process Communication (IPC) primitives.

In the benchmarks tested, the ported kernel consistently performs within a few percent of the hand-coded kernel. This is especially significant in that the setup used was essentially the most convenient to port to: few alterations were made to improve the performance of the relational implementation of Fiasco. In particular, we have not taken advantage of RelC's auto-tuner in this work, which chooses an optimal decomposition for a provided benchmark.

3.4 Verification

We are interested in proving the memory safety of the ported Fiasco kernel. A proof of memory safety certifies the absence of certain memory errors, in particular null dereferences, uninitialized memory usage, and out-of-bounds array accesses, among others.

Compass is an analysis tool capable of proving the memory safety of programs. It does so by tracking all memory accesses throughout a program and generating all of the possible patterns of access. If any one of these possible uses of the program memory could lead to a memory error according to built-in models for general memory errors, Compass reports a possible error. Traditionally, such systems have suffered from high false positive rates, but Compass uses precise reasoning for containers to greatly reduce this, increasing the utility of such analysis tools.

The precise container reasoning used by Compass is a form of static analysis; it does not require that the program to be analyzed be executed in order to prove memory safety, and we thus are not bound by the constraints mentioned in Section 4.2 that restricted our use of standard C++ features or libraries [1].

Compass' ability to understand the particular containers used in programs rests on annotations provided in container implementations. Compass' authors, to perform analysis on sample systems, provided Compass with an annotated set of STL containers already available. Thus, if the Fiasco code could be made to depend on STL containers instead of custom-built data structure primitives, the analysis could proceed without requiring manual data structure annotations. Here, the separation of implementation and interface in data representation synthesis was again useful, because we could replace the Fiasco custom data structure primitives with STL containers without modifying container-manipulating code in the kernel.

Thus, before data analysis could be performed, data structure decompositions were re-written to use STL containers and containers were correspondingly regenerated by RelC. Because Compass does not understand Fiasco's dialect of C++, the codebase was then run through Fiasco's preprocessor to generate standard C++. Compass then was provided the source code for the kernel and the code was analyzed to prove memory safety. Compass reported no warnings during analysis of the kernel.

Note that we have only concerned ourselves here with demonstrating memory safety; higher level reasoning about correct submodule behavior – such as scheduling or memory system – would require further analysis.

4. RELATED WORK

4.1 Operating System Verification

Several other projects have sought to verify microkernel code. In particular, NICTA's seL4 was the first operating system kernel to be completely formally verified [8]. seL4 was custom-built for the verification project. In parallel with the development of the kernel, NICTA developed a complete high-level specification of the behavior of the kernel in the theorem prover Isabelle/HOL. seL4 built on work on the EROS and Coyotos kernels, and a small number of other verified operating systems exist in both academic and industrial sectors [7]. The VFiasco project is perhaps most similar of these projects to the work presented here in that it attempts to verify Fiasco [4]. However, VFiasco focuses in particular on proving high level propeties of correct kernel subsystem behavior.

Our work differs from the approaches in these systems in that we focus on demonstrating the feasibility of building a system suitable for automatic verification of memory safety instead of building an operating system with verified higher-order properties for use in critical embedded applications.

4.2 Automatic Verification

Automatic software verification is an active field of research in programming systems and many promising approaches are under study. Type Qualifiers present one approach, where qualifiers – inferred or programmer-supplied – carry information about expected program behavior and potential security risks [2]. Type Qualifiers have been applied to verify pointer safety in operating system code; one study by Johnson and Wagner used Type Qualifiers to find pointer bugs in Linux kernel code [5].

Work by Kim and colleagues has focused on verification of specific pointer-based data structures, particularly linked data structures [6]. This work indicates the possibility of reasoning about pointer data structures without porting code, but it is not fully automatic and does not come with the advantages besides verifiability – in modularity, implementation independence, and potentially performance – of developing with relations.

These and similar automatic verification techniques vary from our approach in that they do not reason about containers and thus are not able to separate interface from implementation in analysis. The ability to make this distinction can greatly simplify and expedite analysis. However, it should be mentioned that we see our approach as complementary to those mentioned above to verify code that is not container-manipulation or to verify the implementation of container interfaces.

5. CONCLUSIONS

We set out to develop a significant proof of concept for the verification of software systems developed using Data Representation Synthesis. We were able to port core structures of the Fiasco microkernel to structures compiled by Data Representation Synthesis, verify performance, and automatically verify the ported kernel using techniques for precise reasoning about containers implemented in Compass.

We consider this work evidence of the effectiveness of developing arbitrary systems with Data Representation Synthesis. Without increasing code complexity or decreasing performance, the use of relations creates automatically verifiable container-manipulating code where analysis-confounding pointer data structure code once stood.

6. REFERENCES

[1] I. Dillig, T. Dillig, and A. Aiken. Precise Reasoning for Programs Using Containers. *PLDI*, 46(1):187–200, 2011.

[2] J. S. Foster, M. Fähndrich, and A. Aiken. A Theory of Type Qualifiers. *PLDI*, 34(5):192–203, 1999.

[3] P. Hawkins, A. Aiken, K. Fisher, and M. Rinard. Data Representation Synthesis. *PLDI*, 47(6):38–49, 2011.

[4] M. Hohmuth, H. Tews, and S. G. Stephens. Applying Source-code Verification to a Microkernel – The VFiasco Project. *Proceedings of the 10th workshop on ACM SIGOPS European Workshop: Beyond the PC*, 2002.

[5] R. Johnson and D. Wagner. Finding User/Kernel Pointer Bugs with Type Inference. *USENIX Security Symposium*, pages 119–134, 2004.

[6] D. Kim and M. C. Rinard. Verification of Semantic Commutativity Conditions and Inverse Operations on Linked Data Structures. *PLDI*, pages 528–541, 2011.

[7] G. Klein. *Operating System Verification – An Overview*, volume 34. Springer, 2009.

[8] G. Klein, K. Elphinstone, G. Heiser, and K. Engelhardt. seL4 : Formal Verification of an OS Kernel. *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles*, 97(1):207–220, 2009.

[9] A. Lackorzynski and A. Warg. Taming Subsystems: Capabilities as Universal Resource Access Control in L4. *Proceedings of the 2nd Workshop on Isolation and Integration in Embedded Systems IIES09*, pages 25–30, 2009.

[10] E. Meijer and G. Bierman. A Co-relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 54(4):49–58, 2011.

[11] K. Zee, V. Kuncak, and M. Rinard. Full Functional Verification of Linked Data Structures. *PLDI*, 43(6):349, May 2008.