Development of Automatically Verifiable Systems using Data Representation Synthesis

Bryce Cronkite-Ratcliff

Department of Computer Science, Stanford University
brycecr@stanford.edu

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification- Formal Methods, Reliability

Keywords automatic verification; data representation synthesis; container reasoning

1. Problem and Motivation

One of the greatest challenges in formal verification has been 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 [9]. This problem has impeded the development of more effective automatic formal analysis tools for general code, and an approach for automatically reasoning about such structures has proved elusive.

While the analysis of general pointer data structures remains difficult, more progress has been made on the related problem of analyzing containers, a restricted class of data structures that permit insertion, query, removal, and iteration operations. These structures are encountered in numerous heavily used libraries, including the C++ STL and Java Collections Framework. 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 [2]. 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) [4]. 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. The choice of data structure implementation is now under compiler control. As a result, the correctness of data structure operations is now by construction.

In this work, we show 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 relatively simple to build and maintain.

SPLASH '13, October 26–31, 2013, Indianapolis, Indiana, USA.
Copyright is held by the owner/author(s).
ACM 978-1-4503-1995-9/13/10.
http://dx.doi.org/10.1145/2508075.2514874

2. Background and Related Work

This work is based on the union of Precise Container Reasoning and Data Representation Synthesis techniques already mentioned, so some further elaboration of these techniques is warranted.

2.1 Precise Container Reasoning

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. Because container interfaces are widely used in standard libraries such as the C++ STL, automatic analysis of container client programs can be decoupled from the potentially more tedious task of verifying the particular container implementation; one implementation analysis can serve for all the clients of the container [2].

Recent work by Dillig, Dillig, & Aiken developed a technique for precise automatic static analysis of container-manipulating programs, which we will refer to as Precise Container Reasoning (PCR) [2]. PCR differs from previous approaches to container analysis in that it separates containers into position-dependent containers (such as stacks, vectors, and 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 [2]. Compass currently supports analysis of C++.

2.2 Data Representation Synthesis

Data Representation Synthesis (DRS) is a technique developed recently by Hawkins and colleagues to provide a means of decoupling the data structure interface from the data structure implementation [4]. 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 ML-like language and generates data structures in C++ or Java that implement the requested relational specification using the supplied decomposition.

DRS has an advantage in the development of verifiable programs: because DRS is capable of generating high-performance encapsulated data structures that can represent arbitrary relations, the need use hand-coded pointer data structures is reduced or eliminated. In particular we can use existing techniques for precise reasoning about containers to reason about DRS-generated relations.

2.3 Related Work

Several other projects have sought to verify microkernel code. In particular, NICTAs seL4 was the first operating system kernel to be completely formally verified [8]. seL4 was custom-built for the verification project and verification was performed by automatically verifying that the kernel code faithfully implemented a manually written formal specification for the kernel behavior. seL4 built on work on the EROS kernel, and a small number of other verified operating systems exist in both academic and industrial sectors. The VFiasco project is perhaps most similar to the work presented here in that it attempts to verify Fiasco [5]. However, VFiasco focuses in particular on proving properties such as whether the internal page-fault handler terminates for all page faults or whether the memory allocator works correctly (in our work we focus on demonstrating memory safety).

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.

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 [3]. 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 [6].

Recent research in path-sensitive data-flow analysis has extended the scope of programs that can be analyzed with data-flow techniques greatly, and has promising applications in automatic compiler optimizations and formal verification [1].

Work by Kim and colleagues has focused on verification of specific pointer-based data structures, particularly linked data structures [7]. 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 modularty, 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. It should be mentioned that we see our approach as complementary to those mentioned above to verify code that is not container-manipulating or to verify container implementations.

3. Approach

Our primary motivation for this work is to demonstrate the automatic verifiability of code using DRS-generated structures. 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 the relational operations as operations on containers, we can use the automatic approach to verifying code that manipulates containers implemented in Compass.

Briefly, we model a relation R as a trie where each non-leaf node at level l is a container mapping an particular value of an attribute a_l in the relation to a set of partial tuples that correspond to the attributes a_i , $l \leq i \leq S_R$ where S_R is the size of the attribute schema of R. Each leaf node is simply a terminal node that indicates the presence of a particular tuple formed by traversing edges from the root of the trie.

With a mechanism for reasoning about relations as containers, we developed a system to demonstrate the efficacy of this approach.

In particular, we removed all the core pointer data structures from the Fiasco.OC microkernel and replaced them with RelC-generated relations. We then benchmarked the result to demonstrate good performance – we see an average performance loss relative to the original kernel of about 4%. Finally, we are currently working with Compass to determine if we can automatically verify security properties of the resulting code.

This work, to our knowledge, represents the first attempt at a validated approach for automatically verifying non-trivial safety properties of complex computer systems that use arbitrary data relationships.

4. Results

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 are currently applying precise container reasoning techniques to determine the verifiabiliy of the code.

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

Acknowledgments

Thank you to Alex Aiken, Peter Hawkins, Tom & Isil Dillig, and Mooly Sagiv for guidance and support.

References

[1] I. Dillig, T. Dillig, and A. Aiken. Sound, Complete and Scalable Path-sensitive Analysis. *PLDI*, 43(6):270–280, 2008.

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

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

[4] P. Hawkins, A. Aiken, K. Fisher, and M. Rinard. Data Representation Synthesis. *PLDI*, 47(6):38–49, 2011. ISSN 03621340.

[5] M. Hohmuth, H. Tews, and S. G. Stephens. Applying Source-code Verification to a Microkernel The VFiasco Project. *ACM SIGOPS European Workshop: Beyond the PC*, 2002.

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

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

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

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