Contract Discovery from Black-Box Components

Vaibhav Sharma
University of Minnesota
Minneapolis, MN, USA
vaibhav@umn.edu

Taejoon Byun
University of Minnesota
Minneapolis, MN, USA
taejoon@umn.edu

Stephen McCamant
University of Minnesota
Minneapolis, MN, USA
mccamant@cs.umn.edu

Sanjai Rayadurgam
University of Minnesota
Minneapolis, MN, USA
rsanjai@cs.umn.edu

Mats P. E. Heimdahl
University of Minnesota
Minneapolis, MN, USA
heimdahl@cs.umn.edu

ABSTRACT

Complex computer-controlled systems are commonly constructed in a *middle-out* fashion where existing subsystems and available components have a significant influence on system architecture and drive design decisions. During system design, the architect must verify that the components, put together as specified in the architecture, will achieve the desired system behavior. This typically leads to further design modifications or adjustments to requirements triggering another iteration of the design-verify cycle. For software components that are acquired from third-parties, often the only definitive source of information about the component's system-relevant behavior – its *contract* – is its object code. We posit that existing static and dynamic analysis techniques can be used to discover contracts that can help the system designer and specifically discuss how symbolic execution of object code may be particularly well-suited for this purpose.

CCS CONCEPTS

• **Software and its engineering** → *Formal software verification*; *Software reverse engineering*;

KEYWORDS

contract discovery, binary analysis, symbolic execution

ACM Reference Format:
Vaibhav Sharma, Taejoon Byun, Stephen McCamant, Sanjai Rayadurgam, and Mats P. E. Heimdahl. 2018. Contract Discovery from Black-Box Components. In *Proceedings of the 1st ACM SIGSOFT International Workshop on Automated Specification Inference (WASPI '18), November 9, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3278177.3278179

1 INTRODUCTION

Modern computer-controlled systems are often constructed in a middle-out fashion, where available components and existing subsystems often drive system architecture and design to an equal extent as the overall system requirements. Construction of such systems is largely an integration effort focused on putting together available components procured from third-parties to realize system functionality. Software for such components may frequently be supplied only in the form of object-code binaries, accompanied by verification and validation reports providing some evidence of the component meeting its specified claims and intended purposes. It is, however, the system integrator's responsibility to ensure that the component is an appropriate fit for the system context in which it is to be used. While the supplied specifications about the component may be used as a basis to design the system, the integrator must go beyond simply relying on those claims and perform independent assessment of the component in the context of the particular system being constructed. The impact of the component's use on system characteristics such as functionality, safety, security, maintainability and performance is typically assessed by the system integrator.

In this setting, techniques for extracting properties of the component software from its object-code that are relevant to the system context in which it is to be used would be of significant help to system integrators. We refer to such context-specific properties of the component, as the component's contract [2]. The specifications that are provided with the component by its vendor, while useful for this purpose, may neither be necessary nor sufficient for the particular system being built. For example, a temperature sensor that provides readings with no more that 0.1% error in the range of 0-100 °C may be more accurate than needed but specified range may be too narrow for use in a typical weather station for homes in the temperate zone. If the sensor component can be analyzed to determine its operating range and error characteristics, its suitability for a particular application can be reliably determined. In the case of software components, if the binary code can be analyzed to extract relevant contracts, those can then be used to reason about system behavior, e.g., in a compositional fashion as in Assume-Guarantee Reasoning approaches [3]. In this paper, we discuss symbolic execution-based object-code analysis techniques to extract such component contracts.

2 MOTIVATION AND BACKGROUND

Modern computer-controlled systems are often complex and are built by decomposing them into subsystems often created by independent teams or procured from external suppliers. The requirements on the system as a whole are similarly decomposed and allocated to the subsystems. When the subsystems themselves are sufficiently complex, they are further decomposed, leading to an architectural view of the system as a hierarchical organization of interconnected components. This decomposition induces an analysis

WASPI '18, November 9, 2018, Lake Buena Vista, FL, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-6057-9/18/11…$15.00
https://doi.org/10.1145/3278177.3278179

effort to ascertain whether the requirements allocated to the components are sufficient to establish the system-level requirements.

This hierarchical decomposition and the components' requirements do not, however, evolve in a top-down fashion. Existing subsystems and available components significantly influence architectural design decisions and, in turn, requirements allocation. The overall system requirements provide the context for construction around the existing parts, which provides a basis for system design, in a process that is aptly described as middle-out development. What appears as requirements at one layer of the system architecture, would seem to be a design detail from the vantage point of a higher layer in the same architecture [11]. Middle-out development is, in essence, a process to reconcile the desired "whats" and the available "hows". Important to this process is not only what a component is expected to provide—the *guarantees*—but also what a component expects to be true of its context within the system—the *assumptions*. Whether done formally through compositional verification or informally through human reasoning, this process will yield requirements (what guarantees are sought under what given assumptions) that will serve as a basis for contracting out their development or procuring them from third-parties.

To convincingly argue that a system thus built serves its purpose, Hammond et al. developed the notion of a Satisfaction Argument [6], based on Jackson and Zave's World and the Machine model [7]. A satisfaction argument lays out why one should believe that system requirements hold, using a rigorous argument based on the specifications of the system guarantees as well as the assumptions about the system's environment and domain. In the ideal top-down view of development, this would be performed by the system integrator before the subsystems are contracted out for development or procured from third parties. In practice, this is rarely, if ever, the case. The requirements for a subsystem are typically expressed using a combination of artifacts—natural language statements, partial models in notations such as Simulink and Stateflow, prototype code, and at times through implicit appeal to domain and organizational knowledge—which do not easily lend themselves to a rigorous verification of the delivered system. Thus, instead of expecting a formal contract specified *a priori*, we believe that it is necessary and feasible to "discover" those from the artifacts *a posteriori*. A formal satisfaction argument can then be constructed to reason compositionally over the system architecture to determine if the contracts for the sub-systems, working together as defined in the architecture, are sufficient to establish the system's contract (system-level requirements).

Of particular interest, in this paper, are subsystems that are software components supplied by third-parties in object-code format. In this setting, techniques for extracting properties of the component software from its object-code that are relevant to the system context in which it is to be used would be of significant help to system integrators. A range of existing techniques for analyzing object-code, both statically and dynamically, can be applied for this purpose. Specifically, symbolic execution of binary object-code can be used to discover contracts from software components and to construct satisfaction arguments for the system in which the component is used. In cases where the software component matches most but not all of the contract, symbolic execution-based contract discovery can be combined with other techniques to recommend changes to the contract. This paper discusses known applications of symbolic execution which provide confidence that this is indeed feasible and discusses desirable characteristics of such techniques for use as a contract discovery tool. While we do not describe the implementation of a contract discovery tool, this paper provides a foundation for developers attempting to do contract discovery using symbolic execution in the future.

3 APPROACH

In this section we will discuss why binary symbolic execution is an appealing approach for middle-out contract discovery, and compare it to other possible static or dynamic approaches. Next we discuss the previously-studied problem of adaptor synthesis, for which binary symbolic execution has proved effective, and then discuss the problem of automatically suggesting changes to a contract.

3.1 Binary Symbolic Execution

There are several reasons why symbolic execution [8] is well suited for contract discovery and comparison problems. Symbolic execution can operate well even on unstructured code, so it can apply directly to off-the-shelf code available only in object code form. But perhaps the most basic advantage is that symbolic execution already works by translating the behavior of program execution paths into precise formulas, so it is a convenient starting point for other analyses that also reason about program semantics in this way. Contracts are also typically expressed as formulas for component outputs in terms of component inputs, so symbolically executing a component is a convenient first step that expresses its behavior as formulas so that they can be compared with the expectations embedded in a contract.

While symbolic execution produces formulas, its results typically differ from contracts because the formulas are derived from code execution and so describe the implementation's behavior in precise detail. The formulas may be split into many cases if code has complex control-flow structure, and the formula derived from each path will also reflect the exact operations the code performed, perhaps with some syntactic simplification. By contrast, our goal for a contract is a simpler formula which captures the most important aspects of a component's behavior but abstracts over many implementation details. The formulas derived by symbolic execution will typically be too complex to ask developers to read, but if the component matches its contract, the component's behavior, as explored by symbolic execution, should logically imply the contract; in other words, the contract should be a weakening of the component's full behavior. Thus we can use the results of symbolic execution for automated comparison, but when proposing contract changes we should use a more limited grammar of formulas to control the complexity of the contracts that developers reason about.

3.2 Other Static/Dynamic Analyses

Some of the other benefits of symbolic execution for binary contract discovery can be summarized in comparison to other purely dynamic or static approaches.

Symbolic execution of a full component in its concrete execution context shares with purely dynamic analysis the advantage of reflecting the component's behavior precisely, without abstraction: it need not overapproximate the code's behavior, which might cause important behavioral properties to be lost. For such precise analyses, the key challenge is getting sufficient coverage of component behavior. Behavioral coverage is a weakness of purely dynamic

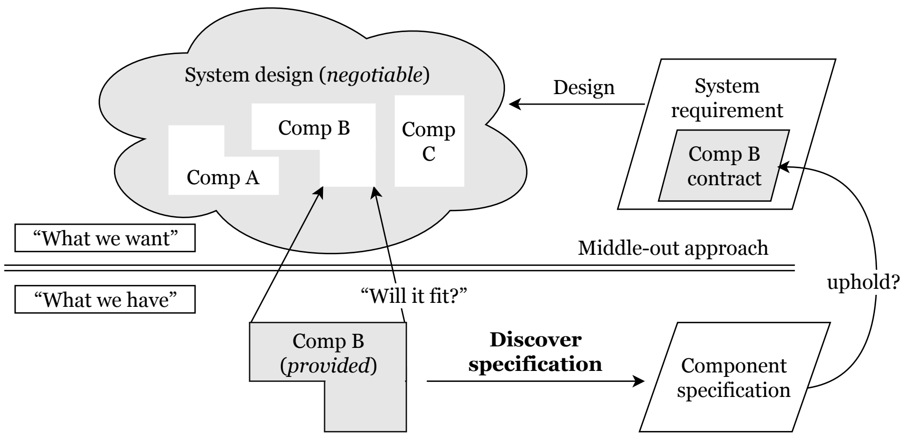

Figure 1: Middle-out Approach

techniques, such as those based on a user-supplied set of test cases, because they cannot exclude the possibility that new behaviors might occur on untested execution paths. Behavioral coverage can be a challenge for symbolic execution as well: if a component has many possible execution paths, it may take a long time to explore them all. But a single symbolic execution path can cover the behavior of a number of concrete test cases, and when uncovered paths remain, the branch conditions produced by symbolic execution provide a logical characterization of the untested inputs.

Because it is desirable to have complete behavioral coverage when reasoning about high-assurance systems, it is natural to consider sound static analyses that can provide such a coverage guarantee in a limited analysis run time. Unfortunately, sound static analysis of unannotated binaries is challenging because of their lack of structural information, which makes it difficult to build tools with such an approach. The two key challenges are the unstructured nature of control flow and memory accesses when analyzing a binary. Binary level indirect jumps and indirect memory accesses take an arbitrary machine word as their addresses, so considered in isolation, an indirect jump might transfer control to any code in a program, and an indirect memory access might read or write from any data structure. The analogous constructs in source-level languages, like function and data pointers, are constrained by the source-level type system, but these types are erased in the translation to binary code and so cannot be relied upon for analysis. Any sound analysis of binaries for another purpose, such as contract discovery, must perform a sound control-flow and data pointer analysis as a prerequisite; note also that the control and pointer analysis are intertwined when code pointers are stored in dynamically allocated memory. These problems are not fundamentally unsolvable, but they are a challenging obstacle especially if a scalable analysis is required. In particular there is a shortage of sound and scalable binary static analysis frameworks that are freely available for research.

3.3 Adaptor Synthesis

An example from previous work of an application of binary symbolic execution for reasoning about component behavior is Sharma et al.'s *adaptor synthesis* [9]. Adaptor synthesis reasons about the behavior of two components using a counter-example guided synthesis loop to determine if one can be wrapped to serve as a replacement for the other. Specifically, adaptor synthesis finds the correct adaptor that can adapt the interface of a function, `f2`, to the interface of a different function, `f1`, such that the adapted `f2` serves as a replacement to `f1`. This relationship is represented as `f1` $\leftarrow$ `f2`. Adaptor synthesis can be applied for discovering the contract of a binary function, `f1`, by finding another function, `f2`, that is adaptably equivalent to `f1`. If a contract is available or can be written for `f2`, then a contract can be written for adapted `f2` because an adaptor only performs interface-level changes such as argument substitution and type conversion. Because adaptor synthesis finds `f2` to be adaptably equivalent to `f1`, the contract for `f2` must be adaptably equivalent to `f1` as well. Thus, adaptor synthesis allows us to discover the contract for a binary function `f1` by using another function, `f2`, for which a contract is available. Sharma et al.'s adaptor synthesis approach uses binary symbolic execution for both checking the equivalence of functions and for synthesizing adaptors. Though Sharma et al. only consider adaptors between two implementations, the same approach can be used to adapt an implementation to match a contract.

```c
int isalpha_component(int c) {
  return table[c] & 1024;
}
int adapted_isalpha(int c) {
  return (isalpha_component(c) != 0) ? 1 : 0;
}
```

Listing 1: Glibc implementation of the *isalpha* predicate and a wrapper around the glibc implementation that is equivalent to the specification in Listing 2

As a simple but concrete example, consider the source code for an `isalpha_component` shown in Listing 1. The `isalpha_component` (which is same as the function `glibc_isalpha` function used by Sharma et al.) checks a bitmask in a previously populated `table` array to see if the bit at position 10 is set. If this bit is set, it signals that the character argument is alphabetic. If this `isalpha` predicate is satisfied, it returns a value of 1024, else it returns 0. The return value of 1024 satisfies the loose contract from the C standard that a non-zero return value be returned if the argument is alphabetic. The specification in Listing 2 is a stronger one that many programmers would think of first, where a true result is represented by the integer 1. In order for `isalpha_component` to match the specification in Listing 2, a return value of 1024 returned by `isalpha_component` must be changed to 1. The `adapted_isalpha` function makes this possible by adapting the return value of `isalpha_component`. (Astute readers may notice another mismatch between this component and contract, which we will address below.)

```
(declare-fun c () (_ BitVec 32))
(declare-fun ret () (_ BitVec 32))
(assert
 (= ret
    (ite
     (or (and (bvuge c #x00000041)
              (bvule c #x0000005a))
         (and (bvuge c #x00000061)
              (bvule c #x0000007a)))
     #x00000001 #x00000000)))
```

Listing 2: *isalpha* predicate in SMT-LIB notation

3.4 Recommending Contract Changes

If the behavior of a binary component, for instance as discovered using symbolic execution, matches the component contract assumed in the system level design, then the component and system are compatible without further changes. But when the component and the system architecture are developed independently, we expect it will be more common that initially the implementation and contract will not match. We envision that an automated system will assist developers in this situation by recommending possible changes to the component contract that would resolve the incompatibility.

One common situation will be that the binary component requires a stronger precondition that was not mentioned in the existing architecture. For instance, returning to the simple `isalpha` example introduced above in Listing 1, suppose that the table-based `glibc` implementation is to be used in a larger system, and the existing component contract does not provide any range limitation on the argument `c`. The implementation therefore does not obey the contract: values of `c` that are too large or small will cause it to give meaningless results, or even crash. But analyzing the relationship between the component and the contract, we imagine that a tool may be able to suggest that if the precondition on `c` is strengthened to $0 \leq c \leq 127$, the component would satisfy this more restrictive contract. It will still be up to the system developer to determine whether this change is consistent with the plan for the rest of the architecture, and it may need changes to contracts of other components or of the system as a whole. For instance, if the component was intended to operate only on ASCII text, the `isalpha` component is suitable, but the character set requirement needs to be propagated to other parts of the system that operate on text. Or, if the system developer decides that the system should support Unicode, the component will need a more complex adaptor or may not be usable at all.

One approach to determining a recommended precondition weakening is to treat the task as a formula synthesis problem: the goal is to find a simple formula that, among possible inputs to the component, distinguishes inputs for which the component's output matches the contract postconditions from inputs that lead to incorrect outputs. A syntax-guided approach is natural because simple and human-comprehensible predicates are preferred, and a counterexample-guided algorithm can be used by selecting a representative set of component inputs that should satisfy or not satisfy the synthesized precondition. As an optimization, one can start by proposing conditions using a fast fixed-grammar tool such as Daikon [5] (in Daikon terminology, a division of samples is called a "splitting condition" [4]). If no suitable condition lies in Daikon's grammar, the analysis can switch to a more expensive synthesis approach that supports an arbitrary grammar [1, 10].

4 CONCLUSION

Complex systems are often built using binary-only black-box components, so the ability to discover or refine contracts for such components will be important to facilitate assurance in these large composed systems. Specifically, in the context of middle-out development, we have identified that a key need is to adjust an in-development system architecture to accommodate a third-party component. We propose using symbolic execution and synthesis to suggest modifications to a component contract to match a binary implementation; given other recent successes in the use of binary symbolic execution, we believe this approach holds significant promise. This short paper has only sketched the outlines of this approach; we hope this will be just the start of a research program that leads to this promising capability becoming a practical reality.

ACKNOWLEDGMENTS

This material is based on work supported by the National Science Foundation under Grant Number 1563920.

REFERENCES

[1] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design (FMCAD)*. 1–8.

[2] A. Beugnard, J. M. Jezequel, N. Plouzeau, and D. Watkins. 1999. Making components contract aware. *Computer* 32, 7 (1999), 38–45.

[3] Darren Cofer, Andrew Gacek, Steven Miller, Michael W. Whalen, Brian LaValley, and Lui Sha. 2012. Compositional Verification of Architectural Models. In *Proceedings of the 4th International Conference on NASA Formal Methods (NFM'12)*. Springer-Verlag, Berlin, Heidelberg, 126–140.

[4] Nii Dodoo, Lee Lin, and Michael D. Ernst. 2003. *Selecting, refining, and evaluating predicates for program analysis.* Technical Report MIT-LCS-TR-914. MIT Laboratory for Computer Science, Cambridge, MA.

[5] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Science of Computer Prog.* 69, 1 (2007), 35–45.

[6] J. Hammond, R. Rawlings, and A. Hall. 2001. Will it work? [Requirements engineering]. In *Requirements Engineering, 2001. Proceedings. Fifth IEEE International Symposium on.* 102 –109. https://doi.org/10.1109/ISRE.2001.948549

[7] Michael Jackson and Pamela Zave. 1995. Deriving Specifications from Requirements: An Example. In *Proceedings of the Seventeenth International Conference on Software Engineering (ICSE'95)*. 15–24.

[8] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (July 1976), 385–394. https://doi.org/10.1145/360248.360252

[9] Vaibhav Sharma, Kesha Hietala, and Stephen McCamant. 2018. Finding Substitutable Binary Code for Reverse Engineering by Synthesizing Adapters. In *11th IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 150–160. http://doi.ieeecomputersociety.org/10.1109/ICST.2018.00024

[10] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. 2006. Combinatorial sketching for finite programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 404–415.

[11] Michael W. Whalen, Andrew Gacek, Darren D. Cofer, Anitha Murugesan, Mats Per Erik Heimdahl, and Sanjai Rayadurgam. 2013. Your "What" Is My "How": Iteration and Hierarchy in System Design. *IEEE Software* 30, 2 (2013), 54–60. https://doi.org/10.1109/MS.2012.173