Component Recovery, Protocol Recovery and Validation in Bauhaus

Rainer Koschke, Yan Zhang

www.bauhaus-stuttgart.de
{koschke, zhangyan}@informatik.uni-stuttgart.de

Abstract

Bauhaus is a research collaboration between the department for programming languages and compilers at the University of Stuttgart and the Fraunhofer institute for experimental software engineering in Kaiserslautern. At last year's Bad Honnef workshop [2], we have outlined future research topics of Stuttgart's Bauhaus group. This year, we summarize the achievements of the last 12 months and elaborate our research directions in more detail. This paper specifically addresses continued research in component recovery based on previous work [7] that additionally leverages our new infrastructure for control and data flow analyses. The paper introduces also relatively new research to recover protocols for the identified components.

1. Improved Component Recovery via Static Control and Data Flow Analyses

A **component** is a computational unit of a system. Components consist of an interface, which offers the resources (types, variables, subprograms) of the component, and the implementation of these resources. The resources of the component coherently contribute to the purpose of the component. An **interface** has a syntactic part that declares the resources provided by the component and a semantic part that describes how the component is to be used correctly. Any possible use of a resource provided by the interface is said to be an **operation** of the component. Operations, hence, range from subprogram calls, variable accesses, instance creations to accessing individual record components of types provided by the component.

Component recovery has been studied extensively within Bauhaus [7]. The overall result is that current techniques that use coarse information about the relationships among types, variables, and subprograms fall short of needed precision. Leveraging our new infrastructure for control and data flow analyses we want develop better techniques based on more fine-grained information. An avenue, for instance, is to refine the so-called accessor classification approach by Würthner and Girard [8], in which subprograms are classified into one or more of the following classes (while Würthner and Girard have just looked at the subprogram signature to decide these cases, we are planning to analyze the subprogram body as described as hints in the following):

- **constructor**: the subprograms creates a new (instance of the) component (hints: call to memory allocation routines like *malloc*, setting record components with literals)
- **destructor**: releases an existing (instance of the) component (hints: call to memory deallocation routines like *free*)
- **modifier operation**: changes the state of an existing (instance of the) component (hints: the data of the component are changed)
- **accessor operation**: returns information about an existing (instance of the) component without changing it (hints: the data of the component are not changed)

A subprogram can play different roles with respect to different types, and the classification could be used to assign the subprogram to the type for which it plays a closer related role according to the following priority: constructor/destructor > modifier > accessor.

Program slicing techniques and points-to analyses for function pointer for more acurate call graphs may be an avenue to come to finer-grained analyses with more reliability. Moreover, component recovery can be combined with other architectural research in Bauhaus, such as feature location and connector recovery [9] in order to identify subsystems.

2. Protocol Validation

During design, a system architect decomposes a larger software system into smaller and more manageable components. For each component, he or she defines an interface of exported declarations. Likewise, by semi-automatic component recovery, we may be able to identify the components and, given these, the externally used resources of the components can be identified as the provided interface. However, semi-automatic component recovery can only identify the syntactic interface. Similarly, in forward engineering, often the semantic part is only described with some informal comments. Consider Figure 1 as an example of a syntactic interface of a component *Stack*.

```
typedef struct {Item contents[100];
                int top; } Stack;

Stack Init (); // constructor

void Push (Stack *S, Item i);
  // pushes i onto S as top element

void Pop (Stack *S);
  // removes top element from S

Item Top (Stack S);
  // return top element of S

int Empty (Stack S);  // true if S is empty

void Release (Stack *S); // destructor
```

Figure 1. Interface of Component Stack.

With only the syntactic interface, it is not clear how the component is to be used correctly. Generally, the exported declarations of an interface entails constraints that cannot

be specified by conventional programming languages. For instance, the subprograms offered in the interface may be subject to certain restrictions of allowed call sequences. If an actual call sequence violates the given restrictions, a failure at run-time may occur. Such failures are often hard to find, because they may become visible only long after the actual fault happened.

For this reason, the allowable way of using a component needs to be explicitly specified. This specification is called the **protocol** of the component. Without protocol specification, the programmer does not know how to use the component. Likewise will a technical auditor without specification not be able to validate the component. If such an specification does not exist or if it is obsolete, it needs to be derived. Hints on the correct usage of the component may be derived from its implementation directly. We call this *glass-box understanding* because the implementation of the component is investigated. Complementary, or even alternatively if the implementation is too difficult to understand or not available, one can also look at the actual usages of the component in – preferably correct – programs. The strategy to derive hints on a component by looking on how it is used without looking at the implementation will be called *black-box understanding*.

2.1. On Static and Dynamic Traces

The actual use of a component may be derived by dynamic or static analysis. For a dynamic derivation, one would prepare use cases that require a certain component, instrument the source or object code, execute the program, and then use a profiler to extract the executed operations of a component as **dynamic traces** for each program run. The advantage of dynamic analysis is that it yields precisely what has been executed. The problem of aliasing, where one does not exactly know at compile time what gets indirectly accessed via an alias, does not occur for dynamic analysis. Moreover, infeasible paths, i.e., program paths for which a static analysis cannot decide that they can never be taken, are excluded by dynamic analysis, too. On the other hand, dynamic analysis lacks from the fact that it yields results only for one given input or usage scenario. In order to find all possible dynamic traces of the component, the use cases have to cover every possible program behavior. However, full coverage is generally impossible because there may be principally endless repetitions of operations.

Static analysis may derive all possible traces - so-called **static traces** - regardless of the actual input. Static traces represent the statically derived potential execution sequences of a component's operations. As operation, we consider any usage of a resource exported by the component, including subprogram calls, access to a global variable of the component's interface, and access to a record component of any type provided by the component. The connection between static and dynamic traces is that each dynamic trace is an instance of a static trace. To put it differently, static traces can be viewed as a grammar and a dynamic trace is a word derived from this grammar.

However, in many cases, static analysis can only safely extract static traces by making conservative assumptions on the program because many questions relevant to traces, like aliasing and infeasible paths, are generally undecidable at compile time. In the view of static traces as a grammar, one may, hence, state that static traces may often be considered a grammar that defines a superset of the actually possible dynamic traces. A static trace that was extracted via infeasible paths or an overestimation of aliasing and that cannot really occur at runtime will be called an **infeasible** static trace in the following. One must also note, that due to the problem of reaching full coverage of all possible inputs is neither feasible in practice, dynamic analysis neither gives the set of all possible dynamic traces. Both static and dynamic analysis are, thus, approximations where static analysis yields the upper bound of all possible traces and dynamic analysis the lower bound.

2.2. Protocol Validation

The purpose of **protocol validation** is to validate that each traces conforms to the specified protocol. Protocols are typically checked at run-time. However, to be on the safe side, one has to validate protocols statically. Static protocol validations has to check that every static trace is either infeasible or is covered by the protocol specification. It goes without saying that protocol validation can only be done semi-automatically since many questions related to protocol validation are generally undecidable. However, it would still be very useful for large systems to at least identify the potential mismatches between static traces and the specified protocol and then let the user decide whether the static traces actually do not conform to the protocol. Again, both static traces and protocols describe a language. Thus, for such a validation, one basically has to show that the language described by the static traces is a subset of the language described by the protocol. Unfortunately, verifying this property is only possible for regular languages in general. Consequently, different authors have proposed to use finite state automata to specify protocols [1, 8]. These protocols express the sequencing constraints on a component's operations only. Constraints on the data passed to the components, for instance, are not part of sequencing constraints. For example, it cannot be expressed with finite state automata that the element that is currently being retrieved from a container component must have been added before.

In order to validate a static trace against a protocol, one can simply carry out the following procedure:

1. represent the static trace and protocol by two deterministic finite state automata, *T* and *P*, respectively,
2. combine these two automata by adding one new starting node, *S*, plus two epsilon transitions from *S* to the starting nodes of *T* and *P*; where the accepting states of the new combined automata is the union of the accepting states of *T* and *P*,
3. minimize the combined automata using Moore's algorithm [5],
4. and check whether every state, *t*, of *T* has at least one equivalent state, *p*, of *P* in the minimized combined automata and, if *t* is an accepting state, *p* is also an accepting state.

An alternative approach was proposed by Olender and

Osterweil, who use a data flow framework in which state transitions are propagated through the control flow graph [8]. The advantage of their approach is that it does not only check universally quantified but also existentially quantified constraints.

2.3. Static Trace Extraction

In order to validate static traces against a protocol, both static traces and protocol must exist. While static traces can be extracted automatically, the protocol needs to be specified by the programmer. However, if the original programmer did not properly specify the protocol, it needs to be derived by someone who might not be familiar with the component. This section describes how we extract static traces. Section 2.4 depicts how protocols can be semi-automatically ascertained using these extracted traces and other information derived from source code.

If the component's interface consists of global variables and subprograms only, deriving the static traces is a simple traversal of the interprocedural and intraprocedural control flow graph that collects the accesses to the global variables belonging to the component and the calls to the subprograms provided by the component in the order in which they occur in the control flow.

If the component exports types, a programmer may create an arbitrary number of values of these types by declaring instances as global or local variables or formal parameters or via dynamically created instances on the heap. In case of instances, the static traces need to be extracted for each instance individually. Actually, these instances are only carriers for values of these types and we are rather interested in the operations executed on the values. For example, a stack instance may be declared, then initialized, some elements may pushed on it, and finally it is passed as an actual parameter to another subprogram. The formal parameter, in turn, that receives this stack value now carries an initialized non-empty stack and, hence, may apply at least one *pop* to it. Consequently, assignments and parameter passing need to be treated properly. If there is a full assignment of *a* to *b*, like "*b* := *a*", *b* inherits the state of *a* and, hence, the static trace of *a* that led to this state. The former value of *a* is overwritten and its lifetime ends. A new lifetime begins for the value that has newly been assigned to *a*, characterized by the inherited static trace of *b*. Note that partial assignments are not treated that way but considered an operation and will be part of the static trace. One may argue that *a.c* := *b.c* for all components, *c*, of *a* and *b* is equivalent to a total assignment *a* := *b*. However, we do not really expect many examples in which a programmer completely assigns a value by enumerating assignments of all parts.

Parameter passing with both value and reference semantics are just special cases of assignments. In order to explicitly represent passing values between carriers as subprogram parameters, we simply link the current static trace of the actual parameter to the formal input parameter and – in case of output parameters – back from the end of the static trace of the formal parameter to the actual parameter. This representation allows maximal sharing for formal parameters of subprograms called more than once. Copying the static traces to and from formal parameters is neither compact nor feasible for cycles in the call graph.

The individual intraprocedural static trace of a local variable or a parameter can be ascertained by a traversal of the control flow graph that collects all operations in which the instance is passed as an argument and all operations that return a value assigned to the instance. For global variables as instances, an analogous traversal of the call graph is needed combined with an intraprocedural traversal of the visited subprograms (excluding operations of the component, of course, as they are considered atomic). Instances can also be introduced as record components of composite variables but they can be handled analogously to regular local or global variables or formal parameters, depending upon the scope of the enclosing variable.

A problem exists for instances that occur in an array. For arrays, we can not decide which elements are really accessed in run-time determined subscripts. For validation purposes, we have to treat the array as an atomic instance and to collect all operations that involve the array itself or any of its elements. Generally, this leads to imprecise results and the maintainer or auditor needs to be notified. For protocol recovery based on extracted static traces, one could also ignore arrays. An even more difficult problem exists for instances created on the heap. Similarly to arrays, we can combine all instances created at a certain heap allocation. The particular heap allocation then serves as an atomic instance. Additionally, we have to track the pointers referring to the instance created on the heap in order to identify operations that involve the instance. Hence, a points-to analysis is needed. Points-to analysis is also required to precisely keep track of references to a value via aliases.

The combination of static traces of all instances and further operations provided by the component make up the static traces for the component as a whole. If there are no dependencies between individual traces of instances and other operations of the component, one can simply unite the static traces. If one cannot exclude dependencies, the static trace for the component as a whole is the sequence of all operations disregarding which instance is passed, which is basically an interleaving of all individual static traces and further operations. For instance, if the stack constructor crashes if a certain limit of instances has been exceeded and, therefore, requires that a certain operation, *may_create*, is called that checks that the number of instance is still below that limit, then any valid static trace is expected to have *may_create* before it constructor call, even though *may_create* has not an argument of type *Stack*.

Our current prototypical static trace extraction is a control-flow oriented traversal that is ignoring aliasing [3]. Only recently, we have finished a points-to analysis based on Wilson's dissertation [10], which we now started to integrate with the static trace extraction. The new precise static trace extraction will be along our static single assignment form and be analogous to interprocedural slicing [6], where only def-use data dependencies of instances and control dependencies are relevant. An interesting result will be to compare the static traces extracted by the different approaches.

2.4. Protocol Recovery

Although research has answered the question on how protocols (limited to sequencing constraints that can be described by finite state automata) can be validated [8], there is no research on how to get these protocols if the original programmer did not specify them – at least to our best knowledge.

Principally, there are three different sources of information in order to find hints on the actual protocol for an undocumented component:

- intra-component information, i.e., the source code of the component itself,
- extra-component information, i.e., how the component is being used,
- and domain knowledge, i.e., how components of a particular domain are typically organized.

Intra-component information may be used to identify dependencies between operations of a component and may help to decide whether individual traces of instances may be considered independent and, hence, need not be collapsed into an interleaved static trace for the component as a whole. We intend to use our side-effect analysis for gathering information on additional dependencies.

Extra-component information is based on the static traces that we extract for uses of the component.

Domain knowledge is needed to relate the operations to the application domain and to understand their semantics. Since our protocol recovery is semi-automatic, domain knowledge is integrated by way of the user who recovers the protocol.

Our method to recover protocols is an iterative interactive process using the extracted static traces as a starting point and unifies them into protocols [4]. The user triggers automatic analyses that identify (potential) opportunities where static traces can be unified and validates them. Since both static traces and protocols are finite state automata according to our point of view and, hence, basically graphs, the unification is a set of graph transformation rules, where the semantics of these transformations can be specified in terms of the underlying language theory for finite state automata. We can identify two alternative transformations:

- semantically preserving transformations, i.e., transformations that do not change the language of the finite state automata,
- and transformations that do change the language of the finite state automata but that could still be allowable.

In the class of semantics-preserving transformations fall conversion of non-deterministic automata into deterministic ones and minimization of the finite state automata by way of Moore's algorithm. If two static traces are not completely equivalent or subsume each other, Moore's algorithm at least identifies common suffixes. A reversed version of Moore's algorithm is also able to identify common prefixes.

In the category of transformations that do not maintain the semantics but that could still be allowable, we can offer, for instance, reordering of operations if they do not have any data dependency according to intra-component data flow analysis. Note the absence of data dependency is not always sufficient to decide whether reordering does not effect the semantics of the program, as exemplified by the following example of two operations, in which *is_initialized* must be called before *is_empty* even though there is no data dependency between these functions:

```
int is_initialized     int is_empty (Stack *s) {
(Stack *s) {             return s->size > 0;
  return s != NULL; }  }
```

Another non-semantics-preserving transformation is, for instance, marking an operation that does not change the state of a component as optional if no other operation is control-dependent on it, which may trigger further transformations. Additional examples can be found in [4].

The user can add information on semantic equivalence of certain operations, opening new opportunities for further graph transformations.

References

[1] Butkevich, S., Renedo, M., Baumgartner, G., and Young, M., 'Compiler and Tool Support for Debugging Object Protocols', Proceedings of the eighth international symposium on Foundations of software engineering for twenty-first century applications, 2000, pp. 50 - 59.

[2] Czeranski, J., Eisenbarth, T., Kienle, H., Koschke, R., and Simon, D., 'Wiedergewinnung von Architekturinformationen: Ausblicke', 2. Workshop Software Reengineering, Fachberichte Informatik, Universität Koblenz-Landau, 8/2000, pp. 21-23.

[3] Hanssen, S., 'Extraktion statischer Traces zur Wiedergewinnung von Protokollen', Studienarbeit Nr. 1768, Institut für Informatik, Universität Stuttgart.

[4] Heiber, T., 'Semi-automatische Herleitung von Komponentenprotokollen aus statischen Verwendungsmustern', Diplomarbeit Nr. 1822, Institut für Informatik, Universität Stuttgart.

[5] Hopcraft, J.E., and Ullman, J.D., 'Introduction to Automata Theory, Languages, and Computation', Addison-Wesley, 1979.

[6] Horwitz, S., Reps, T., Binkley, D., 'Interprocedural slicing using dependence graphs', *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 1, pp. 26-60, January 1990.

[7] Koschke, R., 'Atomic Architectural Component Recovery for Program Understanding and Evolution', Dissertation, Institut für Informatik, Universität Stuttgart, 2000, http://www.informatik.uni-stuttgart.de/ifi/ps/rainer/thesis.

[8] Olender, K.M., and Osterweil, L.J., 'Interprocedural Static Analysis of Sequencing Constraints', ACM Transactions on Software Engineering and Methodology, Vol. 1, No.1, pp. 21-52, January 1992.

[9] Simon, D., and Eisenbarth, T., 'Feature Location and Connector Recovery: New Approaches for Software Understanding', submitted to 3. Workshop Software Reengineering, Bad Honnef, 2001.

[10] Wilson, R.,'Efficient, Context-Sensitive Pointer Analysis for C Programs', Dissertation, Stanford University, USA, 1997.

[11] Girard, J.-F., Würthner, M., 'Evaluating the Accessor Classification Approach to Detect Abstract Data Type', 8th International Workshop on IWPC Program Comprehension, pp 87-95, June 2000