Intra-Module Inference

Shuvendu K. Lahiri[1], Shaz Qadeer[1], Juan P. Galeotti[2], Jan W. Voung[3], and Thomas Wies[4]

[1] Microsoft Research
[2] University of Buenos Aires
[3] University of California, San Diego
[4] École Polytechnique Fédérale de Lausanne

Abstract. Contract-based property checkers hold the potential for precise, scalable, and incremental reasoning. However, it is difficult to apply such checkers to large program modules because they require programmers to provide detailed contracts, including an interface specification, module invariants, and internal specifications. We argue that given a suitably rich assertion language, modest effort suffices to document the interface specification and the module invariants. However, the burden of providing internal specifications is still significant and remains a deterrent to the use of contract-based checkers. Therefore, we consider the problem of *intra-module inference*, which aims to infer annotations for internal procedures and loops, given the interface specification and the module invariants. We provide simple and scalable techniques to search for a broad class of desired internal annotations, comprising quantifiers and Boolean connectives, guided by the module specification. We have validated our ideas by building a prototype verifier and using it to verify several properties on Windows device drivers with zero false alarms and small annotation overhead. These drivers are complex; they contain thousands of lines and use dynamic data structures such as linked lists and arrays. Our technique significantly improves the soundness, precision, and coverage of verification of these programs compared to earlier techniques.

1 Introduction

Program verification is an undecidable problem, which makes it impossible to build automated and precise program verifiers. In the last few decades, research in static analysis and program verification has attempted to improve the precision and scalability of program verification tools without requiring significant user input. Most existing tools for verifying properties of software fall under two extremes: push-button tools based on model checking [7] and abstract interpretation [9] that have little room for user guidance, or contract-based verifiers such as `ESC/Java` [14] and `Spec#` [5] that require the user to specify all the contracts.

Contract-based property checkers hold the potential for precise, scalable, and incremental reasoning. However, it is difficult to apply such checkers to large program modules because they require programmers to provide detailed contracts.

These contracts include an interface specification (preconditions and postconditions for public procedures), module invariants, and internal specifications (loop invariants, preconditions, and postconditions for internal procedures). Manually providing such contracts is infeasible for large modules that typically contain thousands of lines and hundreds of procedures. Operating systems modules such as device drivers, file systems, and memory managers are typically large and consequently remain outside the scope of existing verification techniques.

We argue that given a suitably rich assertion language, modest effort suffices to document the interface specification and the module invariants. This is fortunate because these specifications are the most useful for documentation and program understanding. While the interface specification documents the client's view, the module invariants provide the central argument for establishing the interface specification and other desirable properties of the module. However, the burden of providing internal specifications is still significant and remains a deterrent to the use of contract-based checkers. Therefore, we solve the problem of *intra-module inference*. Given a module with an interface specification, module invariants, and a property to be proved, we infer annotations on loops and internal procedures guided by the provided specifications.

In this work, we demonstrate how to synthesize a broad class of internal annotations containing quantifiers and Boolean structure, in a scalable fashion, guided by the module invariants. Our inference method generates a set of *candidate annotations* using an idea of *exception sets*, and then searches for annotations within the candidate pool using the scalable Houdini algorithm [13]. We formalize our ideas in terms of a general and extensible annotation language comprising *type-state assertions*. The type-state of a pointer can be static or can depend on runtime attributes such as the runtime type of a pointer, values of object fields, and membership (or non-membership) in heap-allocated data structures [22].

We have validated our ideas by building a prototype verifier and using it to verify several properties on Windows device drivers with zero false alarms and small annotation overhead. These drivers are complex; they contain thousands of lines and use dynamic data structures such as linked lists and arrays. We show that proving even simple type-state properties may require tracking type-states related to linked lists and non-trivial aliasing constraints in the module invariants. We then demonstrate how our inference technique is able to infer almost all internal annotations. Our technique significantly improves the soundness, precision and coverage of verification of these programs compared to earlier techniques applied on these programs (see Section 6 for related work).

Our experience leads us to the following conclusions:

1. Having the programmer specify the module invariants and the tool infer the internal annotations is a useful tradeoff in the quest for automated program verifiers that can check general properties with high precision. Given only the property to be proved, inferring the module invariants automatically with reasonable cost seems unlikely because the required invariant may depend on sophisticated type-state abstractions absent from the property. Inference of

internal annotations guided by the structure of the module invariants seems more amenable to cost-effective automation.

2. Searching for internal annotations guided by user-specified module invariants discovers annotations that are understandable by a programmer, unlike intermediate assertions of static analysis tools that can only be understood by machines. This attribute is important if a tool attempts to aid program documentation and incremental checking, in addition to finding bugs.

2 Motivating example

In this section, we show the module invariants and internal annotations required to verify properties of a real-life Windows device driver `kbdclass`. We focus on checking the absence of the following *double-free property*: a pointer of type `DEVICE_OBJECT` that is deleted with a call to `IoDeleteDevice` was allocated via a prior call to `IoCreateDevice`, and an object is not deleted twice. The purpose of this section is to show that the module invariants required to check this property can be expressed succinctly over a set of suitable type-states with a relatively low annotation burden. On the other hand, it is non-trivial to arrive at the relevant invariants mechanically starting from the property of interest, as they contain type-state abstractions, quantifiers and Boolean structure.

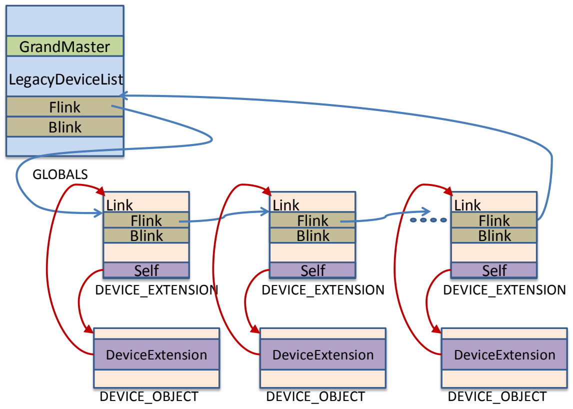

Fig. 1. Data structures in `kbdclass`.

2.1 Module invariants

Module invariants are a set of assertions that are preserved by the public procedures of a module and are strong enough to prove the property of interest.

$$\forall \texttt{x} \in \texttt{MyDevObj} : \texttt{x->DeviceExtension->Self} = \texttt{x} \quad (1)$$
$$\forall \texttt{y} \in \texttt{MyDevExtn} : \texttt{y->Self->DeviceExtension} = \texttt{y} \quad (2)$$
$$\forall \texttt{x} \in \texttt{MyDevObj} : \texttt{x->DeviceExtension} \in \texttt{MyDevExtn} \quad (3)$$
$$\forall \texttt{y} \in \texttt{MyDevExtn} : \texttt{y->Self} \in \texttt{MyDevObj} \quad (4)$$
$$\forall \texttt{z} \in \texttt{GL\_LIST} : \texttt{z} = \texttt{HD} \vee \texttt{ENCL(z)} \in \texttt{MyDevExtn} \quad (5)$$
$$\texttt{HD} \in \texttt{GL\_LIST} \quad (6)$$
$$\texttt{Globals.GrandMaster} \neq 0 \implies \texttt{Globals.GrandMaster} \in \texttt{MyDevExtn} \quad (7)$$
$$\&\texttt{Globals.GrandMaster->Link} \notin \texttt{GL\_LIST} \quad (8)$$

Fig. 2. Module invariants for checking double-free property on `kbdclass`. `ENCL(x)` is a macro for `CONTAINING_RECORD(x, DEVICE_EXTENSION, Link)`.

Figure 2 shows the module invariants required to prove the double-free property on `kbdclass`. Each of these invariants either specifies a property on all pointers satisfying a dynamic type-state, or the type-state of a global variable. In this section, we will explain these invariants with respect to the `kbdclass` module.

Allocation type-states and aliasing invariants. We use the mutable sets `MyDevObj` and `MyDevExtn` to model the allocation (in `IoCreateDevice`) and deletion (in `IoDeleteDevice`) of `DEVICE_OBJECT` pointers in `kbdclass`. The structure of a device object and a device extension are described below (ignore the field of type `PLIST_ENTRY` for now):

```
typedef struct _DEVICE_OBJECT{   typedef struct _DEVICE_EXTENSION{
    void *DeviceExtension;           PLIST_ENTRY    Link;
    ...                              PDEVICE_OBJECT Self;
}                                    ...
DEVICE_OBJECT, *PDEVICE_OBJECT;  }
                                 DEVICE_EXTENSION, *PDEVICE_EXTENSION;
```

Intuitively, each call to `IoCreateDevice` adds a pointer u to `MyDevObj` and a pointer v to `MyDevExtn`, setting u`->DeviceExtension` to v at the same time. Conversely, when a pointer u is passed to `IoDeleteDevice`, it removes u from `MyDevObj` and u`->DeviceExtension` from `MyDevExtn`. The aliasing constraints between the fields `Self` and `DeviceExtension` (see Figure 1) are captured by invariants 1 and 2.

Now, consider the public method `KeyboardPnP` in the `kbdclass` module:

```
requires (DeviceObject ∈ MyDevObj)
NTSTATUS KeyboardPnP(PDEVICE_OBJECT DeviceObject, PIRP Irp){
    ...
    data = (PDEVICE_EXTENSION) DeviceObject->DeviceExtension;
    ...
    case IRP_MN_REMOVE_DEVICE: IoDeleteDevice (data->Self);
    ...
}
```

When the request IRP_MN_REMOVE_DEVICE is passed in the Irp parameter, the procedure deletes the device object DeviceObject . The kernel ensures that KeyboardPnP is only invoked on a device object that was previously allocated by a call to IoCreateDevice in this module. This property is reflected in the precondition given in the **requires** annotation. However, note that IoDeleteDevice is invoked on DeviceObject->DeviceExtension->Self — it is typical of device drivers to extract the device extension from the device object at the beginning and then use the device extension for the rest of the routine. This is safe because DeviceObject->DeviceExtension->Self is identical to DeviceObject, a fact provided by the module invariant 1 in Figure 2.

Type-state for list membership. The kbdclass module also contains a global structure variable Globals of type GLOBALS as shown below:

```
typedef struct _GLOBALS{
    LISTIST_ENTRY  LegacyDeviceList;
    PDEVICE_EXTENSION GrandMaster;
    ...
} GLOBALS;
```

```
typedef struct _LIST_ENTRY{
    PLIST_ENTRY Flink;
    PLIST_ENTRY Blink;
} LIST_ENTRY, *PLIST_ENTRY;
```

The GLOBALS structure contains a pointer to a device extension in GrandMaster, and the head of a list of device extensions in LegacyDeviceList. The LIST_ENTRY structure contains forward and backward links for a linked list; fields of this type represent the head (LegacyDeviceList in GLOBALS) as well as the links (Link in DEVICE_EXTENSION) in a list (refer to Figure 1).

Now consider the public procedure KeyboardClassUnload which deletes all the device objects present in the driver, before unloading the driver.

```
VOID KeyboardClassUnload(PDRIVER_OBJECT DriverObject){
    ...
    // Delete all of our legacy devices
    for (entry = Globals.LegacyDeviceList.Flink;
         entry != &Globals.LegacyDeviceList;){
       PDEVICE_EXTENSION data =
               CONTAINING_RECORD (entry, DEVICE_EXTENSION, Link);
       RemoveEntryList (&data->Link);
       entry = entry->Flink;
       ...
       IoDeleteDevice(data->Self);
    }

    // Delete the grandmaster if it exists
    if (Globals.GrandMaster) {
        data = Globals.GrandMaster;
        Globals.GrandMaster = NULL;
        ...
        IoDeleteDevice(data->Self);
    }
}
```

The loop in the procedure iterates over the list Globals.LegacyDeviceList using the iterator variable entry. The macro CONTAINING_RECORD is used to obtain the enclosing structure given the address of an internal field, and is defined as:

```
#define CONTAINING_RECORD(x, T, f) ((T *)((int)(x) - (int)(&((T *)0)->f)))
```

For each iteration, `data` points to the enclosing `DEVICE_EXTENSION` structure of `entry`. This entry is removed from the list by `RemoveEntryList` and freed by the call to `IoDeleteDevice` on `data->Self`. Finally, the device object in `Globals.GrandMaster` is freed, after the loop terminates.

We argue about the safety of these operations in the remainder of this section. First, we define two macros that are used in the definition of the module invariants in Figure 2, to refer to the linked list in `Globals.LegacyDeviceList`:

```
#define HD       &Globals.LegacyDeviceList
#define GL_LIST  Btwn(Flink, HD->Flink, HD)
```

Here `HD` denotes the dummy head of the linked list and `GL_LIST` denotes the pointers in the linked list. The set constructor $\texttt{Btwn}(\texttt{f}, x, y)$ [20] denotes the set of pointers $\{x, x\texttt{->f}, x\texttt{->f->f}, \ldots, y\}$ when x reaches `y` using `f`, or $\{\}$ otherwise.

1. We need to specify that for each entry `z` of the list apart from the dummy head `HD`, the pointer `CONTAINING_RECORD(z, DEVICE_EXTENSION, Link)` is a `MyDevExtn` pointer. This is specified in invariant 5. Invariant 6 denotes that the set is non-empty. Notice that invariant 5 corresponds to a type-state invariant, where the type-state is determined by membership in a data structure.
2. For any `MyDevExtn` pointer x, x`->Self` should be a `MyDevObj` pointer, to enable the call to `IoDeleteDevice` to succeed (invariant 4).
3. Furthermore, we need to ensure that `IoDeleteDevice` is not invoked on the same device object pointer in two different iterations of the loop. This can happen if x`->Self` $=$ y`->Self`, where x and y correspond to the values of `data` in two different loop iterations. This can be prevented by requiring that any `MyDevExtn` pointer x satisfies x`->Self->DeviceExtension` $= x$ (invariant 2).
4. Finally, invariant 7 ensures that `Globals.GrandMaster` is a valid device extension and invariant 8 ensures that `Globals.GrandMaster` is not present in the `GL_LIST`, and therefore `Globals.GrandMaster->Self` was not freed in the loop.

2.2 Internal annotations

Let us now look at some of the annotations required on the internal procedures to verify the absence of the double-free property modularly. In the ideal case, the module invariants would act as the necessary preconditions, postconditions and loop invariants for all the procedures. However, these invariants can be temporarily broken due to *allocation* of new objects satisfying a given type-state or *mutation* of fields in objects. For instance, many invariants on the global state (invariants 5, 7, 8) do not hold on internal procedures and loops called within the initialization procedure of the module. For `kbdclass`, most of the internal annotations *in addition* to the module invariants can be categorized into one of the following:

Module invariant exceptions. In the procedure `KeyboardClassUnload` below, a device object `*ClassDeviceObject` is initialized after being created by `IoCreateDevice`.

```
NTSTATUS KbdCreateClassObject(..., PDEVICE_OBJECT  *ClassDeviceObject)
{
    ...
    status = IoCreateDevice(..., ClassDeviceObject);
    ...
    deviceExtension =
              (PDEVICE_EXTENSION) (*ClassDeviceObject)->DeviceExtension;
    deviceExtension->Self =  *ClassDeviceObject;
    ...
}
```

Any procedure call between the call to `IoCreateDevice` and the setting of `Self` would violate the invariant 1. In this case, the type-state invariant 1 holds for all pointers with the *exception* of `*ClassDeviceObject`.

Type-states of pointers and their fields. We have to propagate the type-states of pointers available from preconditions of public procedures to internal procedures to utilize the type-state invariants. In another case, we need the loop invariant `entry` $\in$ `GL_LIST` for the loop in `KeyboardClassUnload` (described in Section 2.1); it describes the type-state of the local variable `entry`.

Modifies clauses. In the absence of any annotation about the *side-effect* (or *modifies set*), a procedure havocs (scrambles) all the globals the fields and type-states of pointers in the heap. Modifies annotations are needed to provide precise updates to the relevant fields and type-states.

Conditional annotations. Sometimes the annotations in each of above categories need to be guarded by some conditions (e.g. return value, flags in some parameters). The procedure `KbdCreateClassObject` (encountered earlier in this section) creates and returns a `MyDevObj` pointer through the out-parameter `*ClassDeviceObject`, *only* when the return value is non-negative.

In the remainder of the paper, we formalize the notion of *type-states* (Section 3), and provide mechanisms to generate internal annotations guided by the module invariants (Section 4), and finally evaluate our technique on a set of real-world device drivers (Section 5).

3 Type-state assertions

In this section, we formally define a *type-state* and *type-state assertions* that comprise our annotation language. A *type-state* is a set of pointers, and the type-state of a pointer is determined by membership or exclusion from the set.

$$
\begin{array}{lll}
\phi \in \mathit{Formula} & ::= & \forall \mathtt{x} \in S :\ \phi \mid \psi \\
\psi \in \mathit{BFormula} & ::= & t \in S \mid t \leq t \mid \neg\psi \mid \psi \vee \psi \\
t \in \mathit{Expr} & ::= & \mathtt{x} \mid t\texttt{->f} \mid \&t \mid\ *t \\
S \in \mathit{SetExpr} & ::= & \{\} \mid \mathsf{int} \mid S \cup S \mid S \setminus S \mid \mathtt{Btwn}(\mathtt{f}, t, t) \mid \mathtt{Array}(t, t, t) \mid\ \ldots
\end{array}
$$

Fig. 3. Grammar for type-state invariants. Here `f` refers to a program field. The top-level invariant is of the form ϕ, S is a set expression, and `int` is the set of all integers. We require that `x` not appear in S in the formula $\forall \mathtt{x} \in S :\ \phi$.

Type-states are a generalization of static types in a program and are useful for two purposes: (a) for establishing non-aliasing between pointers belonging to disjoint type-states, and (b) to state a property about a set of pointers satisfying a type-state.

The language of type-states is extensible by the user and includes the following:

1. **Static:** The static types in programs correspond to immutable type-states.
2. **Dynamic:** Some type-states are mutable at runtime. For example, the type-state `MyDevObj` is mutated by calls to `IoCreateDevice` and `IoDeleteDevice`.
3. **Membership in data structures:** As we have seen, we often need to distinguish pointers based on their membership in a list (e.g., `GL_LIST`), a tree or an array. We use set constructors to refer to all elements of a list between two pointers (`Btwn`) or all elements in the range between two pointers or integers [6].

Often, there are interesting sub-typing relationships among type-states. For example, the type-state `MyDevObj` is a subtype of `PDEVICE_OBJECT`.

Figure 3 shows the recursive structure of a type-state assertion. *Formula* represents a top-level assertion, that can either be a quantified type-state assertion, or a *BFormula*. A *BFormula* represents a Boolean combination over type-states and arithmetic relations over terms; *Expr* represents the various pointer terms that can be constructed from a variable `x`. The vocabulary of type-states *SetExpr* is extensible by the user. Here $\mathtt{Array}(a, s, n) \doteq \{a + s * i \mid 0 \leq i < n\ \}$ is the set constructor for all pointers in an array of size n starting at a, where each entry has a size of s bytes.

The quantified assertions represent *type-state invariants* which state a property for all pointers satisfying a given type-state; examples of such an assertion are invariants 1 and 5 in Figure 2. Quantifier-free assertions are used to describe type-states of variables or their fields; invariants 7 and 8 are examples of it. Type-state invariants can be nested: the following invariant establishes the parent-child relationship for all elements in a list:

$$\forall \mathtt{x} \in \mathtt{TypeA} :\ \forall \mathtt{y} \in \mathtt{Btwn}(\mathtt{Flink}, \mathtt{x\text{->}first}, \mathtt{NULL}) :\ \mathtt{y\text{->}parent} = \mathtt{x}$$

4 Generating internal annotations

Given the interface specification and the module invariants, the next step is to infer the internal annotations. There are two broad classes of annotations that need to be inferred for modular verification—assertions and modifies clauses. Assertions include loop invariants and preconditions and postconditions on internal procedures.

As we illustrated in Section 2, sound verification of real-world modules requires module invariants to contain quantifiers and Boolean connectives. Automatic inference of such annotations is challenging and existing procedures for constructing quantified invariants with Boolean structure is limited to relatively small programs [15, 19, 17, 24]. In this section, we demonstrate the use of *exception sets* to synthesize a broad class of internal annotations containing quantifiers and Boolean structure, in a scalable fashion guided by the module invariants.

We use the Houdini [13] algorithm for inferring the internal annotations of a module decorated with module invariants. The algorithm is simple yet scalable: the algorithm takes as input a set of candidate annotations for the internal procedures and outputs the largest subset of the candidate annotations that is mutually consistent. The algorithm initially assumes all the candidate annotations and then greedily removes an annotation that does not hold during a modular verification step — the process terminates when the set of annotations is consistent or empty. Although this can be seen as solving a restricted case of the predicate abstraction problem (often called *monomial* predicate abstraction), the restricted nature of the problem makes it scalable. For example, when the complexity of deciding formulas in the assertion logic is **Co-NP** complete, the complexity of the predicate abstraction problem is **PSPACE** complete, whereas the complexity of monomial predicate abstraction is **Co-NP** complete [21].

The main challenge in using Houdini is to generate enough candidate annotations to capture a broad class of internal invariants containing quantifiers and Boolean connectives. In the next two sections, we illustrate the use of exception sets to generate candidate annotations for assertions (Section 4.1) and modifies clauses (Section 4.2) guided by the module invariants. We conclude the section with some description of conditional annotations that our approach does not capture currently (Section 4.3).

4.1 Candidate assertions

We have found the following kinds of assertions suitable for populating the candidate pool for preconditions, postconditions, and loop invariants.

Type-states of pointers. The various type-states mentioned in the module invariants and the interface specification signify the relevant attributes of pointers that should be tracked. For every well-typed path expression in scope and for each relevant type-state, we introduce a candidate assertion stating that the path expression is in the type-state. The declared types of variables and fields are of great value in reducing the set of pointers to check for membership in a given type-state. These candidate facts are important for instantiating the universally-quantified type-state invariants in the module invariants.

Module invariant exceptions. Consider a module invariant $\forall \mathtt{x} \in S : \ \psi(\mathtt{x})$ which specifies some property $\psi(x)$ holds for all pointers x satisfying S. Such an assertion could be broken at a few pointers, which we call *module invariant exceptions.* We guess that this can happen only for the sets of pointers $\Theta_1, \Theta_2, \ldots, \Theta_k$. We can generate the following candidate assertions:

$$\begin{array}{l} \forall \mathtt{x} \in S : \ \mathtt{x} \in \Theta_1 \vee \ldots \vee \mathtt{x} \in \Theta_k \vee \psi(\mathtt{x}) \\ \forall \mathtt{x} \in \Theta_1 : \ \psi(\mathtt{x}) \\ \ldots \\ \forall \mathtt{x} \in \Theta_k : \ \psi(\mathtt{x}) \end{array}$$

Assuming the different sets $\Theta_1, \ldots, \Theta_k$ are pairwise disjoint, the above candidate assertions allow us to capture the *tighest* exception to the module invariant in terms of the input sets. For instance, if the module invariant holds, then all the candidate assertions hold. On the other hand, if the exceptions to the module invariant cannot be captured in the input sets, then the first assertion would fail. When a set $\Theta_i \doteq \{\theta_i\}$ is a singleton, then the quantified fact $\forall \mathtt{x} \in \Theta_i : \ \psi(\mathtt{x})$ can be simplied to $\psi(\theta_i)$.

Example 1. Consider a simple example with the following module invariant in the “steady state” (recall the set constructor `Array` from Section 3):

$$\forall \mathtt{x} \in \mathtt{Array}(\mathtt{a}, 4, \mathtt{n}) : \ \mathtt{x\text{->}d} = 42$$

For a loop that initializes the array, the loop invariant is a combination of a module invariant exception and a type-state assertion on the pointer `iter`, where `iter` is the iterator over the array.

$$\begin{array}{l} \forall \mathtt{x} \in \mathtt{Array}(\mathtt{a}, 4, \mathtt{n}) : \ \mathtt{x} \in \mathtt{Array}(\mathtt{a}, 4, \mathtt{n}) \setminus \mathtt{Array}(\mathtt{a}, 4, \mathtt{iter}) \vee x\mathtt{\text{->}d} = 42 \\ \mathtt{iter} \in \mathtt{Array}(0, 1, \mathtt{n} + 1) \end{array}$$

The exception set in this example is $\mathtt{Array}(\mathtt{a}, 4, \mathtt{n}) \setminus \mathtt{Array}(\mathtt{a}, 4, \mathtt{iter})$.

4.2 Candidate modifies clauses

A *candidate* modifies annotation for a field `F` looks as follows:

$$\begin{array}{l} \text{modifies F } \Theta_1 \\ \ldots \\ \text{modifies F } \Theta_k \end{array}$$

This is an annotation that specifies that `F` is possibly modified only at pointers in the set $\bigcup_{1 \leq i \leq k} \Theta_i$. These annotations relate the state of `F` at entry and exit from a procedure, or at entry and the beginning of an arbitrary iteration of a loop. Modifies annotations are crucial for tracking the side-effect of a procedure or a loop on the heap.

We model the memory as a collection of maps, one for each type and each field [8]. Notice that the modifies annotation simply says that the sets Θ_i are exceptions to the assertion stating that the map `F` is preserved at all pointers! Hence, we use an almost identical strategy as the module invariant exceptions. If the user provides the above modifies annotations as candidates, we generate the following candidate annotations:

$$\begin{array}{l}
\forall \mathtt{x} \in \mathsf{int}: \ \mathtt{x} \in \Theta_1 \vee \ldots \vee \mathtt{x} \in \Theta_k \vee \mathtt{F}(\mathtt{x}) = \mathtt{old}(\mathtt{F})(\mathtt{x}) \\
\forall \mathtt{x} \in \Theta_1: \ \mathtt{F}(\mathtt{x}) = \mathtt{old}(\mathtt{F})(\mathtt{x}) \\
\ldots \\
\forall \mathtt{x} \in \Theta_k: \ \mathtt{F}(\mathtt{x}) = \mathtt{old}(\mathtt{F})(\mathtt{x})
\end{array}$$

Here `old(F)` refers to the value of the map `F` at the pre-state (either at the entry of a procedure or a loop). For singleton $\Theta_i = \{\theta_i\}$, the quantified assertions can be simplified to $\mathtt{F}(\theta_i) = \mathtt{old}(\mathtt{F})(\theta_i)$.

The modifies clauses for mutable type-states (such as `MyDevExtn`) are dealt with similarly.

4.3 Conditional annotations

In some cases, the above candidate assertions may need to be guarded by the type-states of a few pointers. For example, the type-state of a pointer might depend on the type-state of the return variable and/or some flag in the parameters. In other cases, different type-states of a pointer may be correlated (e.g., invariant 7 in Figure 2). We currently require the user to provide the conditional annotations.

5 Implementation and results

We have implemented a prototype tool for the problem of intra-module inference. The input to the tool is a module (a single compilation unit) written in C, along with the specifications of external procedures that are called from the module, and the interface specifications for the public procedures. The user then describes a set of module invariants (similar to Figure 2) for the module. Our tool generates a set of candidate annotations and infers the internal annotations over them using the Houdini algorithm.

Generation of the candidate annotations from the module invariants requires providing a set of pointers at procedure and loop boundaries. These pointers form parts of type-state assertions or exception sets for module invariants and modifies clauses. Pointers can range over the variables (procedure parameters, globals) in scope and field dereferences over them. However, manually specifying the pointers for each procedure and loop can be cumbersome. To relieve this burden, we provide *patterns* that match against the variables in scope at procedure and loop boundaries to generate actual pointer expressions. For example, to generate pointers of type `PDEVICE_EXTENSION`, we provide a pattern to match against variables of type PDEVICE_EXTENSION, or DeviceExtension fields of PDEVICE_OBJECT variables.

We use HAVOC to translate [8] an annotated C program into an intermediate language BoogiePL [12] and use the Boogie [4] verification condition generator to translate a BoogiePL program into an logical formula. We use an efficient implementation of the Houdini algorithm [13] using the Satisfiability Modulo Theory (SMT) solver Z3 [11].

5.1 Benchmarks

We have evaluated our prototype on 4 sample device driver modules in the Windows operating system. These drivers are distributed with the Windows Driver Development Kit (DDK) [1]. Among these drivers, kbdclass is a class driver for keyboard devices installed in a system, mouclass is a class driver for all mouse devices installed in a system, flpydisk is a class floppy driver, and mouser is a serial mouse driver. The size and complexity of the drivers are mentioned in Figure 4.

For each driver, we check two properties:

1. The *double-free* property, as illustrated in Section 2.
2. The *lock-usage* property, as described below.

The lock-usage property states that all KSPINLOCK locks alternate between *acquired* and *released* states by calls to KeAcquireSpinLock and KeReleaseSpinLock respectively, after the lock has been initialized into the released state by a call to KeInitializeSpinLock. We use a mutable type-state Released to capture the state of a lock; this type-state is modified by these Ke*SpinLock procedures.

Since these locks appear as fields of device extensions, we also augmented the module invariants with the following invariant:

$$\forall \texttt{x} \in \texttt{MyDevExtn} : \bigwedge_i \texttt{Released}\ (\&\ \texttt{x->lock}_i) \qquad \text{(L)}$$

where $\texttt{lock}_i$ is the name of the i^{th} field of type KSPINLOCK. This invariant signifies that at entry and exit from a module, the locks in the module are released.

5.2 Results

Figure 4 summarizes the results of our experiments. The experiments were performed on a 2.4GHz machine running Windows with 2GB of memory. For each driver, we first check the double free property (-df extension) and then check both the properties (-all extension). For checking the -all properties, the additional type-state invariant ("Type") in the module invariant column refers to the invariant L described for the locks. Since flpydisk and mouser do not have any global variables relevant to the properties, their module invariants do not require any type-state assertions on the globals.

The results show that our technique is able to infer most of the internal annotations — the number of manual annotations is small. Besides, the number

Example	LOC	# Pr	# Pub	# Loops
kbdclass	7242	51	28	20
mouclass	6857	50	27	18
flpydisk	6771	35	11	24
mouser	7415	67	27	12

Example	Module Inv		Infrd	Manual			Time	
	Type	Glob		Pr	Cond	Oth	Inf	Chk
kbdclass-df	5	3	1476	3	1	2	480	190
kbdclass-all	6	3	1591	3	4	2	818	228
mouclass-df	5	3	1391	3	1	2	491	185
mouclass-all	6	3	1502	3	4	2	892	273
flpydisk-df	4	0	1355	0	0	0	632	129
flpydisk-all	5	0	1431	0	0	0	827	167
mouser-df	4	0	1608	0	0	0	571	126
mouser-all	5	0	1774	2	0	2	224	124

Fig. 4. Benchmarks and results: "df" extensions check double-free and "all" checks both the double-free and lock-usage property. "Module Inv" is the number of module invariants, and comprises of type-state invariants ("Type") type-state assertions on globals ("Glob"). "Infrd" and "Manual" represent sizes of inferred, and manual annotations. "Pr" is the number of procedures manually annotated. The manual annotations are broken up into conditional ("Cond") or others ("Oth"). "Time" is the runtime in seconds — "Inf" is the time for inference, and "Chk" is the time to check the annotated program.

of inferred annotation is much larger than the set of module invariants, thereby justifying the intra-module inference problem. Finally, our inference technique is scalable; the time taken to generate the inferred annotations ("Inf") is of the same order as the time taken to check the final set of annotations ("Chk"). A distributed implementation of the algorithm would further reduce the time.

We have already seen that the burden of writing the module invariants is low, and these annotations are quite succinct. In fact, most of the module invariants for `MyDevObj` and `MyDevExtn` are reusable across all drivers of a given class, modulo renaming of fields. This further amortizes the annotation effort across a class of drivers, and also provides module invariants that drivers of a given class should document and export.

Some internal annotations still had to be provided manually. For `kbdclass`, most of the manual annotations were on a procedure `KbdCreateClassObject` which is a wrapper around `IoCreateDevice` with partial initialization of some of the fields. These annotations were guarded postconditions, where the guard is a predicate on the return value signifying successful device creation. More such annotations were needed for checking the lock-usage property, as only a subset of the locks were initialized in the procedure violating the module invariant L temporarily on some locks. The other manual annotations required for this module included the loop invariant `entry` $\in$ `GL_LIST` for the `KeyboardClassUnload`.[5] For `mouser`, the additional annotations for checking both the properties come from the need to specify preconditions on the objects of asynchronous procedure calls that are never called directly — these procedures get enqueued along with their input argument (a `DEVICE_EXTENSION` object), and can fire later. For

[5] This loop invariant matches our template for candidate assertions, but our tool currently does not support instrumenting local variables. Therefore, the user has to provide loop invariants involving local variables.

these procedures, we had to specify that the argument satisfied the type-state `MyDevExtn`.

For `mouser`, Figure 4 shows a surprising result: the runtime of the inference component of `mouser-all` is substantially smaller than `mouser-df`, even though the latter checks only one property. Recall that the verification of `mouser-all` required extra manully-provided preconditions on asynchronous procedure calls. Even though the verification of `mouser-df` did not require these preconditions, their absence caused many candidate annotations to be refuted in Houdini and thus the algorithm took longer to converge.

The verification of `kbdclass` and `mouclass` relies on an assumption that has not been mechanically verified. These drivers have linked lists of two different types, `DEVICE_EXTENSION` and `IRP`, that use the same underlying `Flink` field. We assume that linked lists of these two types are disjoint. We have verified this assumption manually, and are currently working on verifying it mechanically.

6 Related work

In this work, we have shown how to extend the precise reasoning performed by modular checkers to large modules in systems code with small annotation overhead. We have shown that verification of simple type-state properties may need invariants about type-states and data structures that can be best specified by users at the level of a module. In this section, we briefly compare with related work on checking type state properties on large modules.

Automated software verification tools for simple type-state properties (e.g. lock-usage, resource leaks) are largely based on predicate abstraction [16] (e.g. `SLAM` [3], `BLAST` [18]), data-flow analysis (e.g. `ESP` [10], `Saturn` [2]) and more recently on interpolants [23]. Most of these approaches lose precision when the analysis requires complex type-state invariants in the presence of open modules or unbounded heap, as shown in Figure 2, resulting in false alarms. Existing automated tools deal with this problem in two ways. First, post processing of the set of warnings is done to heuristically report a subset of warnings whereby real bugs may be hidden. Second, a *harness* is created that nondeterministically calls a public procedure after initializing the heap with a small number of objects, thus avoiding the need to specify quantified invariants on an unbounded set of heap objects. Both of these approaches reduce false alarms by sacrificing soundness.

Tools based on abstract interpretation [9] have been specialized to perform *shape analysis* [25] for systems code. These tools have specialized abstractions to deal with recursive data structures. Recent work using separation logic has been shown to scale to realistic device drivers [26]. However, such tools cannot be used for checking generic type-state properties without building specialized abstract domains. Unlike our method, these tools do not allow the specification and verification of the module invariants.

Verification of object-oriented programs makes heavy use of class or module invariants [4]. However, the focus on annotation inference and automation is secondary — these techniques are primarily focused on scalability of *inter* *module* analysis, where each module has manageable complexity. However, for OS modules with several hundred and possibly thousands of internal procedures, the problem of intra-module inference is crucial for utilizing module invariants. The work on `HOB` [22] is closest in spirit to our work, where a combination of user-specified type-state annotations and inference is used to verify non-trivial programs; however unlike our approach the inference does not leverage the module invariants.

Acknowledgments

We wish to thank Aditya Nori, Julien Vanegue, and the anonymous reviewers for their detailed comments.

References

1. Windows Driver Kit. http://www.microsoft.com/whdc/devtools/wdk/default.mspx.
2. A. Aiken, S. Bugrara, I. Dillig, T. Dillig, B. Hackett, and P. Hawkins. An overview of the Saturn project. In *Program Analysis for Software Tools and Engineering (PASTE '07)*, pages 43–48, 2007.
3. T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Programming Language Design and Implementation (PLDI '01)*, pages 203–213, 2001.
4. M. Barnett and K. R. M. Leino. Weakest-precondition of unstructured programs. In *Program Analysis For Software Tools and Engineering (PASTE '05)*, pages 82–87. ACM, 2005.
5. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *Construction and Analysis of Safe, Secure and Interoperable Smart Devices*, LNCS 3362, pages 49–69, 2005.
6. S. Chatterjee, S. K. Lahiri, S. Qadeer, and Z. Rakamarić. A reachability predicate for analyzing low-level software. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '07)*, LNCS 4424, pages 19–33, 2007.
7. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 2000.
8. J. Condit, B. Hackett, S. K. Lahiri, and S. Qadeer. Unifying type checking and property checking for low-level code. In *Principles of Programming Languages (POPL '09)*, pages 302–314, 2009.
9. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for the static analysis of programs by construction or approximation of fixpoints. In *Symposium on Principles of Programming Languages (POPL '77)*, pages 238–252, 1977.
10. M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *Programming Language Design and Implementation (PLDI '02)*, pages 57–68, 2002.
11. L. de Moura and N. Bjorner. Efficient incremental e-matching for SMT solvers. In *Conference on Automated Deduction (CADE '07)*, LNCS 4603, pages 183–198, 2007.
12. R. DeLine and K. R. M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, 2005.

13. C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. In *International Symposium of Formal Methods Europe (FME '01)*, pages 500–517, 2001.
14. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Programming Language Design and Implementation (PLDI'02)*, pages 234–245, 2002.
15. C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *Principles of Programming Languages (POPL '02)*, pages 191–202. ACM Press, 2002.
16. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *Computer-Aided Verification (CAV '97)*, LNCS 1254, pages 72–83, 1997.
17. S. Gulwani, B. McCloskey, and A. Tiwari. Lifting abstract interpreters to quantified logical domains. In *Principles of Programming Languages (POPL '08)*, pages 235–246, 2008.
18. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Principles of Programming Languages (POPL '02)*, pages 58–70, 2002.
19. S. K. Lahiri and R. E. Bryant. Constructing quantified invariants via predicate abstraction. In *Verification, Model Checking and Abstract Interpretation (VMCAI '04)*, LNCS 2937, pages 267–281, 2004.
20. S. K. Lahiri and S. Qadeer. Back to the future: revisiting precise program verification using SMT solvers. In *Principles of Programming Languages (POPL '08)*, pages 171–182, 2008.
21. S. K. Lahiri and S. Qadeer. Complexity and algorithms for monomial and clausal predicate abstraction. Technical Report MSR-TR-2009-2012, Microsoft Research, 2009.
22. P. Lam, V. Kuncak, and M. C. Rinard. Generalized typestate checking for data structure consistency. In *Verification, Model checking, and Abstract Interpretation (VMCAI '05)*, LNCS 3385, pages 430–447, 2005.
23. K. L. McMillan. Lazy abstraction with interpolants. In *Computer Aided Verification (CAV '06)*, LNCS 4144, pages 123–136, 2006.
24. K. L. McMillan. Quantified invariant generation using an interpolating saturation prover. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '08)*, LNCS 4963, pages 413–427, 2008.
25. S. Sagiv, T. W. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems (TOPLAS '98)*, 20(1):1–50, 1998.
26. H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. W. O'Hearn. Scalable shape analysis for systems code. In *Computer Aided Verification (CAV '08)*, LNCS 5123, pages 385–398, 2008.