Considering Typestate Verification for Quantified Event Automata

DOI:
10.1007/978-3-319-47166-2_33

Document Version
Accepted author manuscript

Considering Typestate Verification for Quantified Event Automata

Giles Reger

University of Manchester, Manchester, UK

Abstract. This paper discusses how the existing static analyses developed for typestate properties may be extended to a more expressive class of properties expressible by a specification formalism originally developed for runtime verification. The notion of typestate was introduced as a refinement of the notion of type and captures the allowed operations in certain contexts (states) as a subset of those operations allowed on the type. Typestates therefore represent per-object safety properties. There exist effective static analysis techniques for checking typestate properties and this has been an area of research since typestates were first introduced in 1986. It has already been observed that common properties monitored in runtime verification activities take the form of typestate properties. Additionally, the notion of typestate has been extended to reflect the more expressive properties seen in this area and additional static and dynamic analyses have been introduced. This paper considers a highly expressive specification language for runtime verification, quantified event automata, and discusses how these could be viewed as typestate properties and if/how the static analysis techniques could be updated accordingly. The details have not been worked out yet and are not presented, this is intended for later work.

1 Introduction

This paper describes preliminary work considering the relationship between the typestate verification static analysis technique and a specification language for dynamic analysis (runtime verification). There are two main motivations behind this work:

1. Such static analyses can be used to reduce the amount of work required at runtime by partially evaluating properties (as shown in previous work [8, 27]); and
2. The considered specification language (originally for runtime verification) can express properties not currently considered for static analysis and extending these analyses could strengthen such techniques.

Typestate properties [34] typically take the form of finite state machines attached to single types. This is also a common form of specification in runtime verification [14] and the relationship between the two has been explored previously [1, 8].

However, runtime verification typically considers more expressive properties such as non-safety properties, quantification over multiple objects, arbitrary state and existential quantification. The extension to multi-object typestates is must common in runtime verification (e.g. in the JavaMOP work [26]) and has already been explored [8,

27] in the context of typestate analysis. In this work I consider the expressive specification language of quantified event automata (QEA) [4] that captures all of the above mentioned extensions. This is based on the parametric trace slicing [11] approach but introduces additional language features taking it far beyond what has currently been considered in typestate analysis.

This paper considers how QEA can be related to typestate and how the related static analysis techniques from typestate verification could be extended to support these more expressive properties.

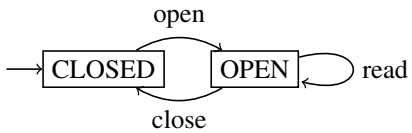
Scope. I restrict my attention to typestate verification and review the relevant related topics in the next section. Notably I have not yet looked at *dependent types* [9] which seem heavily related (this relation is touched on in [13, 25]). Although gradual typing is discussed briefly. I have also (so far) omitted extensions of JML with temporal constraints [20, 22, 35]. It seems likely that including these topics, and other automata-based verification techniques (e.g. [15]), will shed further light on possible ways to combine static and dynamic analysis for quantified event automata.

Structure. I begin by reviewing typestate verification (Section 2) and the quantified event automata specification language (Section 3). I then discuss the extensions of typestate required to support QEA and the possible static analysis extensions to support this (Section 4). I conclude with a discussion of plans for the future (Section 5).

2 A Review of Typestate Verification

Typestate properties [34] were introduced as a programming language concept. Whilst the initial work did not consider a language with objects, the notion has become associated with allowable operations on objects. In general, whilst types restrict the operations that can be performed on an object of that type, there may be different contexts in which only a subset of those operations should be allowed. Types are extended with a notion of state where only certain operations are allowed in each state. Each object of a type has a typestate with some operations updating the state i.e. a typestate property describes a finite state machine.

For example, the following typestate for a File type only allows `open` operations in the CLOSED state and allows `read` and `close` operations in the OPEN state with the `open` and `close` operations changing the state.



Typestate properties are safety properties i.e. they define behaviour that should always happen and can be violated by a finite trace. Importantly, the language described be a typestate property is *prefix closed*. A consequence of this is that a violation has a single witness in the code i.e. an instruction that causes failure.

At this point it is probably worth pointing out that there are two approaches to typestate verification in the literature:

1. The largest body of work focuses on developing new type systems and programming language concepts
2. Other work focusses on existing languages and programs and considers adding typestate support to these

Whilst more work has been done in the first area I am more interested in the second.

2.1 Typestate Verification

Verification of typestate properties is straightforward in the sense that one only needs to construct the *control-flow graph* (CFG) of the program and track each instance of an object. Here an instance of an object is introduced wherever an object is created in the code and that instance is identified by the variable it was assigned to. For certain classes of typestate properties and programs there exist polynomial time algorithms for verification [16]. For example, if a program is shallow (pointers are single-level i.e. allocated objects may not contain pointers) and the property is omission-closed (omitting an event from a valid trace gives a valid trace) then the problem can be reduced to a reachability problem over a graph of polynomial size (as in the IFDS framework [31]).

However, this process becomes non-trivial in the presence of *aliasing* and in the case of single-object typestates (see below for an alternative) most of the effort is concerned with the aliasing issue. The issue is that when objects can be aliased it is not possible to track a single typestate per object any more. Instead it is necessary to track an *abstract object* referring to possible objects and their possible states. The soundness of such techniques depends on the precision of the approach used to disambiguate pointer references. Note that (in general) typestate verification is undecidable in the presence of recursive data structures [23, 28]; a further motivation for combination with dynamic techniques.

There are two general approaches to pointer analysis: *alias analysis* [3] and *points-to* analysis [2, 33]. Alias analysis computes a set of pairs of variables that may or must point to the same location. Similarly, points-to analysis computes, for each pointer, the (points-to) set of variables that p may or must point to. Clearly these are similar techniques. To be useful, approaches generally take the form of *whole-program analysis* (i.e. interprocedural analysis) which requires the complete source code. Precision then depends on whether it is *context* and *flow* sensitive i.e. if it considers call-points or instruction order. Fink et al. utilise a context and flow sensitive analysis for typestate verification [17]. Requiring the full source code can be restrictive and whole-program analysis can be expensive. An alternative, modular, approach is to restrict or annotate aliasing to provide the analysis with enough information to reason about aliased objects effectively. For example, Bierhoff and Aldrich add a notion of *access permissions* [7] to typestates.

An additional concern is *subtyping* i.e. if a type has an associated typestate what should we require of its subtypes. This is dealt with by the notion of behavioural subtyping [24] which dictates the allowed behaviours of subtypes. This is most easily dealt

```
1  interface Iterator<C : Collection, k : Fract> {
2    states available, end refine alive
3
4    boolean hasNext() :
5      pure(this) ⊸ ((result = true ⊗ pure(this) in available)
6                    ⊗ (result = false ⊗ pure(this) in end))
7    Object next():
8      full(this) in available ⊸ full(this)
9
10   void finalize():
11     unique(this) ⊸ immutable(c,k)
12 }
13
14 interface Collection {
15   void add(Object o) : full(this) ⊸ full(this)
16   int size() : pure(this) ⊸ result ≥ 0 ⊗ pure(this)
17   // remove(), contains() etc similar
18
19   Iterator<this,k> iterator():
20     immutable(this,k) ⊸ unique(result)
21 }
```

Fig. 1. Example of a typestate property with access permissions taken from [7].

with in the case where typestates are built-in to the programming language and there have been various type systems developed to deal with such cases within the context of typestate verification [13, 6]. I am not aware of any work that deals with this issue without introducing a new type system.

2.2 Multi-Object Typestates

Whilst the most common application of typestates remains the augmentation of an object type with a notion of state, it has been observed that it can be useful to define and check *multi-object* typestates. The above property on files only considered objects of File type and was therefore *single-object*. If we want to capture properties of the relationship between objects then necessarily we must refer to multiple objects. For example, the property that before reading from a FileReader object associated with a File we must first check that we have read access to the File. Here there are two objects related by the fact that the FileReader is associated with a particular File. Checking this property not only requires us to track the aliasing of each single object but also the relationships between objects. We will see further examples of these *multi-object* typestates later. There are two approaches to handling multi-object typestates.

The first approach is to keep the single-object view and include predicates on related (referenced) objects. In [7] the concept of *access permissions* are added to typestate to additionally indicate (i) how a reference is allowed to modify the referenced object, and (ii) how the object may be accessed through other references. This concept is, therefore, tightly related to aliasing. Figure 1 gives such a typestate property for the well known *UnsafeIterator* property that states that an iterator created from a collection should not be used after the collection is updated. Here this is achieved by line 20 which says (roughly) that whilst the collection is read-only (immutable) then the iterator is accessible (only) by the given reference (unique). Therefore, if the collection is

```
1  tracematch(Iterator i, DataSource ds){
2    sym create_iter after returning (i):
3        call(Iterator DataSource.iterator()) && target (ds);
4     sym call_next before:
5         call(Object Iterator.next()) && target(i);
6      sym update_source after:
7          call (* DataSource.update(..)) && target(ds);
8
9      create_iter call_next* update_source+ call_next
10     {
11        throw new ConcurrentModificationException();
12     }
13 }
```

Fig. 2. A tracematch property for the *UnsafeIterator* property from [1].

updated then the iterator loses its access permission (cannot be used) thus preventing concurrent modification (the above property). This relates the collection (*this*) and the iterator (*result*) by placing a restriction on the iterator dependent on some property of the collection. Note the two states `available` and `end` on Iterator to indicate when it is safe to call `next` (see lines 5, 6 and 8). Clearly this property captures more than the traditional *UnsafeIterator* property i.e. it restricts usage of the iterator object further.

The second approach is to specify multi-object typestates as separate entities. The only existing formalism for this seems to be that of tracematches [1]. These were first introduced as an extension of the AspectJ AOP system to temporal pointcuts i.e. instead of matching single points in the code a regular expression was given to match sequences of points. The semantics is based on slicing (as described later) and is suffix-matching. Figure 2 gives a tracematch property for the *UnsafeIterator* property described above. Lines 2-7 relate abstract events to specific points in the code and line 9 gives a (suffix-matching) regular expression that captures violation of the property. This is defined separately from the code, with the matching parts of the code used to identify events, following the AOP style. Then the tracematches are *weaved* into the code in a separate compilation step that adds additional instrumentation and inserts the specified code fragment wherever a match occurs.

The two main pieces of work in this second area both consider a setting where a tracematch property will be dynamically checked and static analysis is employed to remove instrumentation points in the code i.e. the property is partially evaluated statically. The Clara [8] work implements a number of increasingly precise analyses for partial evaluation. The most effective analysis is a flow-insensitive analysis that computes the may-point-to sets of variables in each transition statement and removes transition sequences without overlapping sets, as these would not relate to consistent bindings. Naeem and Lhotak [27] introduce an updated (operational) semantics for tracematches making them more suitable for static analysis. They then use this to introduce a technique for alias analysis to allow flow-sensitive tracking of individual objects along control flow paths. The final analysis conceptually tracks tracematch states for combinations of relevant objects (e.g. each pair of distinguishable collection and iterator objects). This is then refined to track pairs of over and under-approximations per object for efficiency reasons.

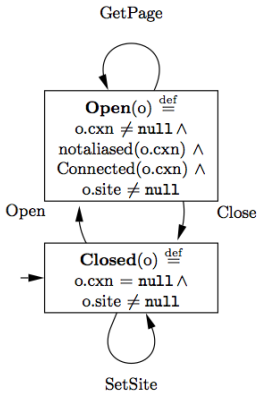


Fig. 3. Illustration of typestate with state invariants labelling states taken from [13].

2.3 State Invariants and Pre/Post Conditions

There is a relationship between the notion of typestate and the usage of invariants, although the original work on typestates did not consider this. It is well known that a combination of object invariants and method pre and post conditions can be used to specify and verify certain kinds of program behaviour. Clearly a method precondition captures information about the required state of the object before the method call and the postcondition captures information about the state of the object after the method call. This highlights the relationship between a concrete notion of 'states' an object can be in and the abstract notion of state in a typestate property; however, it is not clear to this author that there can always be a direct mapping.

In [13] typestates and object invariants are combined in an *object typestate*, a language feature where a typestate is defined in terms of what properties hold of an object's concrete state. A similar approach is taken in [7] where typestates are mapped to predicates on fields. In both pieces of work they note the need for *intermediate states* that exist in the typestate property but do not relate to an existing state invariant i.e. states that should be passed through within a method body.

Figure 3 shows an illustration from [13] where they discuss how typestates of a web page fetcher can be defined in terms of invariants on the fields of that object. Note that they include aliasing information in this invariant.

The example from [7] in Figure 1 shows how they combine logical expressions in their invariants in the following excerpt:

```
1  boolean hasNext() :
2    pure(this) —∘ ((result = true ⊗ pure(this) in available)
3                   ⊗ (result = false ⊗ pure(this) in end))
```

It is not clear how such statements are checked in the analysis.

Additionally, the notion of combining runtime verification of state-based properties with static analysis code annotations has been explored in [12]. Here the temporal

behaviour is checked dynamically whilst the code annotations are checked statically, representing an alternative combination.

2.4 Gradual Typing

There is an area of type theory that deals directly with the notion of mixing static and dynamic analysis: *gradual typing* [32] is the idea that some parts of the program can be statically type-checked whilst other parts are left to be type-checked dynamically (at runtime). This concept has been applied to typestate and there exists a body of work that has now been formulated as *typestate-oriented programming* [18, 36] which describes a Gradual Featherweight Typestate system.

3 Quantified Event Automata

Quantified event automata (QEA) [4] (see also [21, 29]) is a highly expressive specification language with an efficient runtime verification tool MARQ [30] (developed by the author). I refer the reader to previous publications for the technical details. Additionally I will not review the topic of runtime verification and refer the reader to relevant publications (e.g. [14]). In this section I review the fundamental concepts necessary for this paper in an example-led fashion.

3.1 The Structures

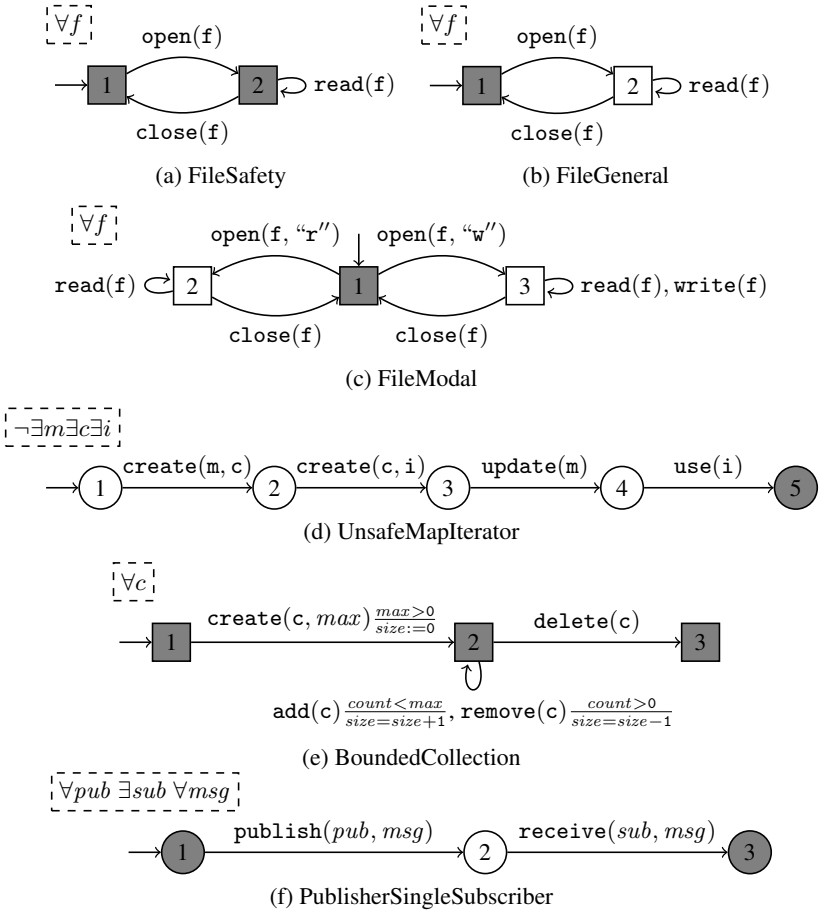
A QEA consists of an *event automata* and a list of *quantifications*. Event automata are an extended form (i.e. with variables) of finite state machine over *data words*. The alphabet of an event automaton consists of *events* built from event names and parameters that are either variables or constants. Additionally, the transitions of an event automaton can include guards (predicates on bindings of variables) and assignments (update functions on bindings of variables). An event automaton is therefore over zero of more variables and the quantifier list may quantify zero or more of these variables.

3.2 Examples

Let us consider some examples which will be used later to describe the semantics of QEA and discuss their role as typestate properties. We will present QEAs graphically and in this notation shaded states are accepting states, square states have a failing completion (if no transition can be taken an error occurs) and circular states have a skipping completion (if no transition can be taken then the event is skipped).

We will use the following properties (illustrated in Figure 4):

a) *FileSafety.* This is a QEA for the property previously used to introduce typestate properties on page 2. A file f can be in two states, open or closed, if closed it can only be opened and if open it can be read or closed.
b) *FileGeneral.* This adds a non-safety element to the previous property. A file that has been opened must eventually be closed.



(a) FileSafety

(b) FileGeneral

(c) FileModal

(d) UnsafeMapIterator

(e) BoundedCollection

(f) PublisherSingleSubscriber

Fig. 4. Quantified event automata examples

c) *FileModal.* This extends the previous file example further; a file can be opened in one of two modes, read or write, and in read mode it cannot be written to.

d) *UnsafeMapIterator.* This concerns unsafe iteration over collections constructed from maps. If a collection c is created from a map m and an iterator i is created from c then if map m is updated the iterator i should no longer be used. This is related to, but not the same as, the previously discussed *UnsafeIterator* property.

e) *BoundedCollection.* On creation a collection has a maximum size max and can contain at most max objects. Additionally, it should not be used after deletion.

f) *PublisherSingleSubscriber.* Every publisher has at least one subscriber that reads all messages published by that publisher.

3.3 Quantification via Parametric Trace Slicing

Quantification is handled via *parametric trace slicing*, a quantification $\forall f$ means that for every value in the domain of f one should consider the (minimal) subtrace mentioning that value, called a *trace slice*. That is, given a binding $[f \mapsto v]$ we ask if the subtrace mentioning only v is accepted by the automaton where f is replaced by v. For multiple quantified variables one should consider the slice for each *combination* of values for the respective variables.

I illustrate this approach using the above *FileSafety* property. Consider the trace

$$\texttt{open(A).open(B).read(A).close(A).open(C).read(B).close(B).open(C)}$$

there are three possible values for f (A,B, and C) and therefore three trace slices

$$\begin{array}{ll} f \mapsto A & \texttt{open(A).read(A).close(A)} \\ f \mapsto B & \texttt{open(B)read(B).close(B)} \\ f \mapsto C & \texttt{open(C).open(C)} \end{array}$$

each trace slice is evaluated on the automaton where f is replaced by the appropriate value. In this case we can see that the slice for $f \mapsto C$ is not accepted by the automaton and therefore the whole trace is not accepted (as the quantification was $\forall$).

In the case of existential quantification (as in the *PublisherSingleSubscriber* property) the semantics is the obvious one; *at least one* trace slice needs to be accepting.

3.4 Event Automata are Extended Finite State Machines

The parametric trace slicing approach can be parameterised by any mechanism for evaluating trace slices. In QEA this mechanism is event automata, which can use variables, guards and assignments to capture highly expressive properties. This is demonstrated in the previous formulation of the *BoundedCollection* property. Two *free* variables are introduced: max to store the maximum size and $size$ to track the current size of the collection. The syntax $\frac{guard}{assignment}$ is used to introduce basic arithmetic predicates and functions. The only non-obvious part of the semantics for these variables is that they are updated whenever they match a value in the trace. For example, when $\texttt{create(c,}\ max\texttt{)}$ matches with the concrete event $\texttt{create(c,5)}$ the value 5 is bound to max.

In MARQ (the runtime verification tool for QEA) arbitary code can be introduced as guards and assignments. This obviously extends expressiveness costing us analysability. Here we stick to basic arithmetic guards and assignments. The extension for more expressive theories is a separate research effort.

3.5 A Finite Trace Semantics with Four Values

Finally, QEA are defined over *finite traces*, which leads to a decision about what to do at the end of the trace (a topic that has received attention previously e.g. [5]). The choice taken here is to use a four-valued semantics. Consider the *FileGeneral* property on the two traces

$$\texttt{read(A).open(A)} \qquad \textit{and} \qquad \texttt{open(A).read(A)}$$

neither trace is correct but they fail for different reasons and we would like to separate these failures. The first trace breaks the safety requirements whilst the second trace does not satisfy the reachability requirement. Notice that the second trace can be *extended* to a good trace but the first cannot. The four possible verdicts are

- *Success.* This trace and all extensions will be accepted
- *Failure.* This trace and all extensions will be rejected
- *Weak Success.* This trace is accepted but some extension may be rejected
- *Weak Failure.* This trace is rejected but some extension may be accepted

Clearly safety properties can only have *Failure* or *Weak Success* verdicts as once violated all extensions will also be violating.

4 Towards Typestate-like Verification for QEA

In this section I reflect on how the quantified event automata introduced in the previous section could be handled (partially in some cases) statically using techniques from typestate verification. As previously discussed, a large amount of work on typestate analysis considers the introduction of new programming language concepts. However, I am interested in the other approach which considers existing programs and programming languages. I make an exception for extensions via additional annotations (e.g. JML) as they do not alter the behaviour of the underlying program.

In the following I discuss possible directions for typestate-like verification for QEA motivated by the examples presented in the previous section.

4.1 Single Object Properties

Clearly the *FileSafety* property is a standard typestate property. However, the QEA does not contain enough information to perform typestate analysis as there is no link between the abstract QEA property and the concrete program. Traditionally, typestates annotate programs like types. But a QEA is a separate object. This is also the case in tracematches, but in that instance the link to the program is built-in i.e. events are specified as pointcuts. However, in QEA the assumption is that some separate instrumentation will create the link between concrete program event and abstract specification event.

There are two alternatives here. One could provide pointcut instrumentation (as in tracematches) and this seems the most natural approach. However, it would also be possible to add annotations to the code that indicate what the event is. This would allow for more fine-grained associations as the AspectJ approach requires events to relate to method calls.

Once this has been sorted then the previously discussed techniques could be used to statically (partially) evaluate a QEA. The partial part is, of course, due to the possibly imprecise nature of reference disambiguation. In a setting where the starting point is dynamic analysis, any imprecision should be dealt with at runtime i.e. if a violation is detected due to an over or under approximation then the runtime checks must be preserved.

4.2 Non-Safety Properties

In the *FileGeneral* property there is a non-safety element. When the typestate is in the open state there are two problematic behaviours to consider:

- The program can be shown to possibly terminate
- The program can be shown to possibly diverge

In either case the bad state is not left and this constitutes a violation. Note that this relates to the finite trace semantics discussed earlier. In QEA it is assumed that a trace is finite, however during static analysis one can consider the possibility of divergence. I discuss the two cases separately.

Early Termination. To detect such errors statically one would need to detect the possibility of (ordinary[1]) termination. As a rather trivial example consider the following piece of code.

```
public void writeSetToFile(Set<Integer> set, String name){
  File file = new File(name);
  file.open();
  for(Integer i : set){
    if(i==0){
       System.out.println(``Error'');
       System.exit(0);
    }
    file.write(i);
   }
   file.close();
}
```

There is a path leading to termination between `open` and `close` and therefore the property is (statically) violated. In the analysis one should label exit points of the program and consider their reachability. The issue is then whether certain paths in the control-flow graph are realisable by real executions. As before, the level of precision achieved will depend on whether the analysis is interprocedural and if it is context-sensitive.

This notion of termination is perhaps strange when considering the standard runtime verification approach. In the runtime verification literature it is (generally) assumed that we detect program termination without knowing which part of the monitored system this termination originates from. Or it is detected by monitoring the exit point of the main method/functional block. Therefore, there is no discussion of using static analysis to remove instrumentation points, as (usually) none are added. In this analysis it may be that the violation occurs in a part of the code that would not be instrumented.

Note that there may be different kinds of termination, for example we may wish to allow exceptional termination. This could be achieved by (automatically) extending the automaton as in Figure 5. Now one must search for paths to termination that do not contain a `close` or `fileException` event.

[1] One cannot reason about abnormal termination such as the machine being switched off!

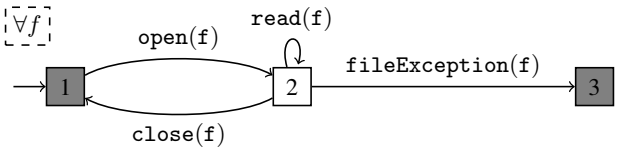


Fig. 5. Adding exceptional termination to *FileGeneral* property.

Divergence. To show that it is possible that the program may never reach the next state transition it would be necessary to establish divergence. For example, via a possibly non-terminating loop. Clearly this is undecidable in general. However, there exists a lot of work [10, 19] on showing that a particular program either always terminates or may possibly never terminate. This particular analysis appears more complex than the previous one and, perhaps, less fruitful.

4.3 Multi Object Properties

The *UnsafeMapIterator* property is a standard case of a multi-object typestate which has been dealt with in previous work. As previously discussed, there are currently two existing approaches [8, 27] to multi-object typestate verification. As discussed in [27], the analyses are complementary. A starting point for multi-object typestate verification for QEA would be to extend either approach, possibly also combining them. As [27] is flow-sensitive and the later discussions demand such an analysis, it would seem that starting with this work would be sensible.

4.4 Guarded Transitions

In the *FileModal* property there is a check on the value of a method call with different values leading to different states. At this point we note that the variables present in the QEA are not necessarily in correspondence with variables in the program (a common misconception) and it might be that the value expected in the QEA would need to be extracted during instrumentation (for example via a method call). In general, it may be necessary to rewrite the program to make such values explicit.

To understand how to statically analyse this property let us consider the following piece of code that satisfies the property.

```
Set values = getValues();
boolean output = values.size() > 0;
File file = output ? new File(name,``w'') : new File(name,``r'');
String line = null;
Set seen = new Set();
while((line = file.readLine()) != null){
  seen.add(line);
}
for(String value : values){
  if(!seen.contains(value)){ file.write(value); }
}
```

Every path containing `write` necessarily starts with `new File(name,``w'')` as if the `values` set has no elements then the iterator containing the `write` will not execute. To check this statically one could carry predicates on program state with the object abstraction i.e. in this case at the point a file is created there are two abstractions that `file` could refer to: a file in state *read-only* with predicate $output = false$ and a file in state *read/write* with a predicate $output = false$. These predicates can be used to determine which paths the abstract object may take, in this case the abstraction where $output = false$ would not enter the final `for` loop. In other words, we could perform some form of symbolic computation. In more complex scenarios (e.g. numeric guards) some form of abstraction would be required.

Where the guard differentiates between valid and invalid behaviour the guard could be added as a precondition or assertion to be checked by standard methods (e.g. deductive verification). But in the general case where future behaviours are determined by the guard, or the property is non-safety, this would not be sufficient.

4.5 Statefull Typestates

In the *BoundedCollection* property there is a need to track and update the values of two specification variables i.e. the typestate has some persistent state. Let us consider the following incorrect code for this property.

```
Collection fill(int value){
  Collection c = new Collection(value);
  for(int i=0;i<=value;i++){
    c.add(i);
  }
  return c;
}
```

There is an out-by-one error in the loop. To check this property we should add the information held by the specification into the code. This could simply be achieved by adding *ghost* variables to track the values in the specification. For example:

```
Collection fill(int value){
  Collection c = new Collection(value);

  //added code
  int max = value;
  int size = 0;

  for(int i=0;i<=value;i++){
    c.add(i);

    //added code
    assert(size+1 < max);
    size++;
  }
  return c;
}
```

Here single variables are added, but in general these would need to exist per object (or collection of objects) i.e. if there were two collections here there would need to be two copies. This could become complicated in the presence of aliasing.

Once these variables are added then the symbolic computation of the previous step could check the guard (added explicitly here).

This example indicates a further complexity of the analysis, often met in static analysis, that of loops. Here, to establish that a violation occurs it would be necessary to establish (automatically) the relationship between the loop counter and the `size` variable and to conclude that `size=max` and `size+1 < max` is inconsistent.

4.6 Existential Quantification

In the *PublisherSingleSubscriber* property we have alternating quantification. Previously, it was necessary for all abstract objects (or collections of objects in the multi-object case) to satisfy the given property. This changes with existential quantification and alternation.

For a single existential quantifier there needs to be a single object across the whole program that satisfies the property. However, this object does not need to satisfy the property on all control paths as each path represents a different execution trace and the requirement is just that there exists an object per execution. Therefore, one object might satisfy it in one control-flow and another in a different one. With multiple quantifiers there is now a relation between the objects and it may be necessary to find an object of one type per an object of another, as in the *PublisherSingleSubscriber* example.

It would seem that some of this could be handled by post-processing of detected violations i.e. analysing which paths contain violations and whether there exists an object with the necessary non-violating paths. But in general it is not clear how to effectively deal with this feature.

5 Conclusion

In this paper I have reviewed typestate verification and the QEA specification language and discussed how the former could be applied to the later. My next step will be to attempt to do this concretely.

As mentioned previously, a starting point will be to take the existing work on multi-object typestate analysis [8, 27] and see if this can be extended. The source code for Clara [8] is available online and there is already support for extending the framework to new tools, and the original authors already did this for JavaMOP. I have obtained the source code for [27] from the authors.

An alternative approach would be to take a tool for (single-object) typestate verification and extend this to add the notions of symbolic computation discussed in the previous section. This is not something I have fully explored yet.

References

1. Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble.

Adding trace matching with free variables to AspectJ. *SIGPLAN Not.*, 40:345–364, October 2005.

2. Lars Ole Andersen. Program analysis and specialization for the c programming language. Technical report, 1994.
3. Andrew W. Appel and Jens Palsberg. *Modern Compiler Implementation in Java*. Cambridge University Press, New York, NY, USA, 2nd edition, 2003.
4. Howard Barringer, Yliès Falcone, Klaus Havelund, Giles Reger, and David E. Rydeheard. Quantified event automata: Towards expressive and efficient runtime monitors. In *FM*, pages 68–84, 2012.
5. Andreas Bauer, Martin Leucker, and Christian Schallhart. The good, the bad, and the ugly, but how ugly is ugly? In *Proceedings of the 7th international conference on Runtime verification*, RV'07, pages 126–138, Berlin, Heidelberg, 2007. Springer-Verlag.
6. Kevin Bierhoff and Jonathan Aldrich. Lightweight object specification with typestates. *SIGSOFT Softw. Eng. Notes*, 30(5):217–226, September 2005.
7. Kevin Bierhoff and Jonathan Aldrich. Modular typestate checking of aliased objects. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 301–320, New York, NY, USA, 2007. ACM.
8. Eric Bodden, Patrick Lam, and Laurie Hendren. Clara: A framework for partially evaluating finite-state runtime monitors ahead of time. In *Proceedings of the First International Conference on Runtime Verification*, RV'10, pages 183–197, Berlin, Heidelberg, 2010. Springer-Verlag.
9. Ana Bove and Peter Dybjer. Language engineering and rigorous software development. chapter Dependent Types at Work, pages 57–99. Springer-Verlag, Berlin, Heidelberg, 2009.
10. Marc Brockschmidt, Byron Cook, Samin Ishtiaq, Heidy Khlaaf, and Nir Piterman. T2 : Temporal property verification. In *Proceedings of TACAS'16*. Springer, January 2016. Full version of TACAS'16 paper.
11. Feng Chen and Grigore Roşu. Parametric trace slicing and monitoring. In *TACAS '09*, pages 246–261, Berlin, Heidelberg, 2009.
12. Jesús Mauricio Chimento, Wolfgang Ahrendt, Gordon J. Pace, and Gerardo Schneider. *Runtime Verification: 6th International Conference, RV 2015, Vienna, Austria, September 22-25, 2015. Proceedings*, chapter StaRVOOrS : A Tool for Combined Static and Runtime Verification of Java, pages 297–305. Springer International Publishing, Cham, 2015.
13. Robert DeLine and Manuel Fähndrich. Typestates for objects. In *ECOOP 2004 — Object-Oriented Programming, 18th European Conference*, volume 3086 of *Lecture Notes in Computer Science*, pages 465–490. Springer Verlag, June 2004.
14. Y. Falcone, K. Havelund, and G. Reger. A tutorial on runtime verification. In Manfred Broy and Doron Peled, editors, *Summer School Marktoberdorf 2012 - Engineering Dependable Software Systems, to appear*. IOS Press, 2013.
15. Azadeh Farzan, Matthias Heizmann, Jochen Hoenicke, Zachary Kincaid, and Andreas Podelski. *Automated Program Verification*, pages 25–46. Springer International Publishing, Cham, 2015.
16. John Field, Deepak Goyal, G. Ramalingam, and Eran Yahav. *Static Analysis: 10th International Symposium, SAS 2003 San Diego, CA, USA, June 11–13, 2003 Proceedings*, chapter Typestate Verification: Abstraction Techniques and Complexity Results, pages 439–462. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.
17. Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. Effective typestate verification in the presence of aliasing. *ACM Trans. Softw. Eng. Methodol.*, 17(2):9:1–9:34, May 2008.
18. Ronald Garcia, Éric Tanter, Roger Wolff, and Jonathan Aldrich. Foundations of typestate-oriented programming. *ACM Trans. Program. Lang. Syst.*, 36(4):12:1–12:44, October 2014.

19. Jürgen Giesl, Frédéric Mesnard, Albert Rubio, René Thiemann, and Johannes Waldmann. *Automated Deduction - CADE-25: 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, chapter Termination Competition (termCOMP 2015), pages 105–108. Springer International Publishing, Cham, 2015.
20. Alain Giorgetti and Julien Groslambert. *JAG: JML Annotation Generation for Verifying Temporal Properties*, pages 373–376. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.
21. Klaus Havelund and Giles Reger. Specification of parametric monitors - quantified event automata versus rule systems. In *Formal Modeling and Verification of Cyber-Physical Systems*, 2015.
22. F. Hussain and G. T. Leavens. temporaljmlc: A jml runtime assertion checker extension for specification and checking of temporal properties. In *2010 8th IEEE International Conference on Software Engineering and Formal Methods*, pages 63–72, Sept 2010.
23. William Landi. Undecidability of static analysis. *ACM Lett. Program. Lang. Syst.*, 1(4):323–337, December 1992.
24. Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, November 1994.
25. Iain McGinniss. *Theoretical and Practical Aspects of Typestate*. PhD thesis, University of Glasgow, 2014.
26. Patrick Meredith, Dongyun Jin, Dennis Griffith, Feng Chen, and Grigore Roşu. An overview of the mop runtime verification framework. *J Software Tools for Technology Transfer*, pages 1–41, 2011.
27. Nomair A. Naeem and Ondrej Lhotak. Typestate-like analysis of multiple interacting objects. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, OOPSLA '08, pages 347–366, New York, NY, USA, 2008. ACM.
28. G. Ramalingam. The undecidability of aliasing. *ACM Trans. Program. Lang. Syst.*, 16(5):1467–1471, September 1994.
29. Giles Reger. *Automata Based Monitoring and Mining of Execution Traces*. PhD thesis, University of Manchester, 2014.
30. Giles Reger, Helena Cuenca Cruz, and David Rydeheard. MarQ: monitoring at runtime with QEA. In *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'15)*, 2015.
31. Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 49–61, New York, NY, USA, 1995. ACM.
32. Jeremy Siek and Walid Taha. Gradual typing for objects. In *Proceedings of the 21st European Conference on ECOOP 2007: Object-Oriented Programming*, ECOOP '07, pages 2–27, Berlin, Heidelberg, 2007. Springer-Verlag.
33. Manu Sridharan and Rastislav Bodík. Refinement-based context-sensitive points-to analysis for java. *SIGPLAN Not.*, 41(6):387–400, June 2006.
34. R E Strom and S Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1):157–171, January 1986.
35. Kerry Trentelman and Marieke Huisman. *Extending JML Specifications with Temporal Logic*, pages 334–348. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.
36. Roger Wolff, Ronald Garcia, Éric Tanter, and Jonathan Aldrich. Gradual typestate. In *Proceedings of the 25th European Conference on Object-oriented Programming*, ECOOP'11, pages 459–483, Berlin, Heidelberg, 2011. Springer-Verlag.