Typestate Checking for Actionscript 3

Yun-En Liu and Qi Shan

December 10, 2010

1 Introduction

This project proposes a compile-time check for function calls in a game system written in Actionscript 3, based on the technique known as "typestate checking". Actionscript 3 is the main language used to write flash games. Most games can be represented internally as a finite state machine, with functions that are valid only in certain states and functions that transition the game to a new state.

Strom and Yemini [1] define typestate checking as, "To track typestate in a program at compile-time, we make typestate a static invariant property of each variable name at each point in the program text. That is, if a variable name has a particular typestate at a particular point in the program text, then the corresponding execution time data-object will have that typestate regardless of the path taken in the program." Thus, typestate checking tries to associate states with variables/objects, according to their types. The compiler stores state information of all variables and allows a certain variable to have different states at different program points. Certain procedures change the typestate. A common use of typestate checking is to make certain at compile time that certain functions are called on objects only when the objects are in valid states.

Consider the case of File I/O. A file is in the state of either "open" or "closed". A file handle can be used to make calls to open or close a file, or to perform read or write operations to a file, while these read and write operations are only valid while the file is open. Generally speaking, function calls are only valid while the object is at a certain state, which implies that only specific orderings of these calls make sense and would give an error-free execution at runtime. For example, writing or reading to a file before opening it, or closing the file a multiple times, are invalid orders of execution. By maintaining a state at compile time for the object indicating whether the file is opened or closed, and checking that functions are only called at appropriate states, the compiler can guarantee that there will be no certain types of I/O exceptions at runtime.

Typestate checks help to prevent run time problems caused by calling a function at an invalid state. Typestate checking allows the programmer to make explicit the states of an object and the legality of different operations in different states. For example, the programmer can explicitly specify, usually at the beginning of the function definition, that a function "close" can only be called when the object state is "open", and changes the state of the object into 'closed". Another potential advantage, which is not implemented in our system, of this analysis is that it allows the compiler to insert memory deallocation code, rather than rely on

the garbage collector to reclaim unused objects.

Extending typestate checking to Actionscript 3 would be useful for detecting invalid function calls on game engines that would otherwise throw errors at runtime. As Actionscript 3 is an object-oriented programming language, in this project, we determine the legality of a certain function call by analyzing the object states, which are defined by programmers, at the corresponding program point. To do this, the compiler keeps tracking the state of each object in the compiling process. Programmers can explicitly describe the state transfer by writing pre- and post-conditions for each function, and declare possible states for each class at the point of class definition. During the semantic analysis, for each function call, the compiler examines the state of the corresponding object and reports compilation error message if the object is not in the state required by the pre-condition. A legal function call will change the state of the object to the post-condition. It is noted that we don't consider aliasing in this project.

In addition, we will add support for typestate checking in Actionscript 3's event-based architecture by ensuring that event listeners that assume objects are in certain typestates may only be active when those objects are in fact in those typestates. Anecdotally, a large fraction of game bugs the authors have encountered have been the result of misfired or dangling event listeners that should not have been active during that particular game state. These bugs are difficult to find and debug, so a compile-time guarantee that these event listener errors will not occur would be very valuable.

2 Typestate annotation examples

In basic typestate analysis, objects have a set of valid typestates, and typestate constraints on functions. Some functions may only be called when the object is in a particular typestate, called a "precondition," and some functions may transition the object to a different typestate, called a "postcondition." It is illegal for a function to be called when its precondition is not satisfied. As a simple example, files may only be read while they are in the "open" typestate, and the only way to shift them from the "closed" to "open" typestate is to call the function open().

In our work, programmers are able to specify valid typestates for any class, as well as the preconditions and postconditions for functions of that class. We allow two types of functions. The first type has a precondition typestate, and one or more postcondition typestates. The second type has no preconditions or postconditions, and does not affect typestate at all. One could imagine more types of functions, such as a function that has postconditions but not preconditions or a function which has different postconditions depending on the preconditions; we leave these additional types of functions as future work.

We extend basic typestate checking with support for event listeners, as well. Game engines in Actionscript 3 are generally controlled through user input, which is captured using events. These event listeners may be triggered at any time as long as they are active, making it difficult for the compiler to provide any kind of guarantee that the function calls made in the event listener are typestate-safe. If the compiler knows when the event listeners are active, however, then it can perform typestate checking. Our approach is to have the programmer associate event listeners with typestates, and whenever a function changes an object's typestate, it must remove all current event listeners and may only add listeners associated with the new typestate.

In this manner, the compiler can assume that event listeners are only ever triggered during their associated typestate.

More specifically, programmers may create event listeners and annotate them with a typestate. They must specify typestate_x() functions, where x is the name of a typestate; these functions may add event listeners associated with typestate x. They must also provide a removeAllEventListeners() function that is guaranteed to remove all event listeners, and call this function at the start of all typestate_x() functions. Thus, whenever typestate_x() is called, it is guaranteed that only listeners associated with x will be active. Finally, functions with postconditions [x, y] must call either typestate_x() or typestate_y() before returning.

As an example of programmer annotations, a programmer might create a ”File” object that can be controlled either programatically or with user input:

```
public class File() {
   typestates("closed", "open");

   public function open():void {
      typestate_transition("closed", "open");
      ...
      typestate_open();
      return;
   }

   public function close():void {
      typestate_transition("open", "closed");
      ...
      typestate_closed();
      return;
   }

   public function read(numBytes:int):String {
      typestate_transition("open", ["closed, "open"]);
      ...
      if (stillHasData()) {
         typestate_open();
         return data;
      }
      typestate_closed();
      return data;
   }

   public function toString():void {
      ...
   }

   //Event listener that can be triggered only during the "open" state
   private function openListener(e:Event):void {
```

```
        typestate_listener("open");
        close();
    }

    //Event listener that can be triggered only during the "closed" state
    private function closedListener(e:Event):void {
        typestate_listener("closed");
        open();
    }

    //User-provided function that removes all event listeners
    private function removeAllEventListeners():void {
        ...
    }

    //Makes certain only "open" listeners are active
    private function typestate_open():void {
        removeAllEventListeners();
        addEventListener(MouseEvent.CLICK, openListener);
    }

    //Makes certain only "closed" listeners are active
    private function typestate_closed():void {
        removeAllEventListeners();
        addEventListener(MouseEvent.CLICK, closedListener);
    }
}
```

The above File object has the typestates ”closed” and ”open.” As ”closed” is the first declared typestate, it will be the default state when a new file is created. The open() and close() functions may only be called when the file is ”closed” or ”open,” respectively, and transitions the file to the other typestate. This transition removes all event listeners and adds the listeners associated with that typestate. The read() function may only be called when the file is in the ”open” state, and may cause the file to become ”open” or ”closed.” Finally, toString() may be called at any time and will not affect the file’s typestate.

Thus, the following example would be a legal usage of the file object:

```
var f:File = new File(); //currently "closed"
print(f.toString());
f.open(); //"closed" -> "open"
print(f.toString());
f.close(); // "open" -> "closed"
print(f.toString());
```

While the following would not be legal:

```
var f:File = new File(); //currently "closed"
var i:int = random();
if (i == 0) {
   f.open(); //"closed" -> "open"
}
else {
   print(f.toString());
}
f.close(); //ERROR: The file might be "closed," if the else branch was taken
```

In Actionscript 3, a more typical use case would look like this:

```
var f:File = new File(); //currently "closed"
addChild(f); //display the file to the screen; user clicks to control it
```

Regular typestate checking is not very helpful in this case. However, our event listener typestate checking allows the compiler to ensure that typestate constraints are not violated - there is no way a file object will call close() while the file is in the "closed" typestate, because the event listener that calls close() is only ever active while the file is in the "open" typestate.

Finally, aliasing is a large problem for typestate checking:

```
var f1:File = new File();
var f2:File = f1;
f1.open();
f2.open();
```

Without alias analysis, the compiler will view this as typestate-safe. However, the call to f1.open() also shifts f2 to the "open" typestate, so in fact this program will throw a typestate error at runtime. We ignore the problem of aliasing as outside the scope of this paper, and do not provide any guarantees if aliasing occurs. Generally speaking, programmers do not need to alias global game engines, so this is not a significant problem for ensuring that a game engine is typestate-safe.

3 Related Work

Strom and Yemini proposed typestate checking in 1986 [1], and argued that it enhances the reliability of a program with compile-time detection of syntactically legal but semantically nonsensical execution sequences.

Whaley et al. tackle the opposite problem of automatically learning a finite state machine from existing code[2]. In their work, they first identify where methods of an object throw exceptions when variables are null or set to some illegal constant, often -1. They then identify which methods may cause those variables to

take on those illegal values, and extract a finite state machine showing which methods may be safely called after which other methods. This information could potentially help programmers understand safe usages of interfaces, as well as provide the compiler with enough information to flag potentially unsafe sequences of method calls.

Applying typestate checking to object oriented languages is a difficult problem, as figuring out the sharing relationships, for example, aliasing of objects references, among different objects is hard at compile time, but trivial at run-time, by comparing the references. DeLine and Fähndrich handled the problem by proposing a program framework of explicitly claiming pre- and post-conditions of the object's methods [3]. This framework extends the typestates checking methods for non-object-oriented and imperative languages. Their framework interprets typestates as predicates over objects, and considers the case of changing typestates through inherited methods in subclasses.

Lam et al. [4] relax previous work that supports only a flat set of object states and requires a limited number of typestate transitions, by proposing a more general reformulation of typestate systems. Unlike most previous work which associates a single typestate with each variable, and determines the legality of a certain function call, the proposed formulation treats each typestate as an abstract set of objects. If an object is in a certain state, it is considered to be a member of the abstract set which corresponds to the typestate. Then it uses object field values to determine the membership of objects to verify abstract set specifications for programs. The reformulation leads to the system which supports a more general class of typestate system concepts, include composite typestates, hierarchical typestates, and cardinality constraints on the number of objects that are in a given typestate.

Aliasing is a problem for typestate checkers. If the compiler cannot determine the underlying object (or reference) pointed by a certain variable at the compile time, as typestate-changing methods are called by those variables, the compiler will not be able to figure out the states of the corresponding objects. For this reason, the first implementations of typestate checking disallowed aliasing [1]. That is not to say typestate checking is impossible in the face of aliasing - for example, Fink et al. and Bierhoff and Aldrich recently combined alias and typestate analyses to allow aliasing [5][6]. Bierhoff and Aldrich use an abstraction called "access permissions" to combine typestate and object aliasing information in their framework. In this work, however, we will ignore aliasing. Programmers of Flash games generally only need one global copy of the game engine in any given program, so lack of alias analysis will not be harmful. Disallowing aliasing has the added benefit of allowing the typestate checking algorithm to run in polynomial time, compared to being PSPACE-hard when arbitrary aliasing is allowed [7].

Most recently, Aldrich et al. propose typestate-oriented programming [8], in which typestate analysis is built into the language itself. They demonstrate this with a Java-like language, Plaid, in which functions must specify whether their arguments may be aliased and what typestates they expect from object parameters. In addition, Saini et al. propose a core calculus for Plaid programming language, named $Plaid_{core}$, in which states and permissions are used to help the typestate checking [9]. In their system, a state is attached to each object as an attribute of that particular object, at compile time. Permissions are used to handle the presence of aliasing.

4 Algorithm

We wrote a dataflow analysis in the Actionscript 3 compiler for typestate checking. The algorithm iteratively computes the typestates each object could be in at every point in the program, given the current best estimate of objects' typestates at all previous points in the program. Once a full iteration is performed without any changes in variable typestate, there is a final pass to ensure that all preconditions are fulfilled, or else the compiler exits with an error.

To be more precise: For each statement, the algorithm will maintain an estimate of which typestate each variable is in at the end of the statement. These estimates take the form of a set of possible typestates. At the start, every local variable at every location begins with an assignment the empty set. Any argument to the function or global variable begins with an assignment of every possible typestate. Of course, one could imagine adding a mechanism for programmers to annotate function arguments or global variables with typestates, but we found it unnecessary when augmenting our game engines with typestates.

The algorithm will now compute a fixed point. There are two cases for the effect a statement has on one variable:

1.) If the statement includes a function call on a variable that has one or more postconditions, this set of postconditions are now assigned to that variable at the end of the statement. This ignores the preconditions for the function entirely, and simply assumes the function call will work - the algorithm will verify later that preconditions are satisfied.

2.) If the statement does not include a function call with postconditions on a variable, the variable's typestate is now the union of the estimated set of typestates of all the predecessors.

In reality, there is a third case where a statement may have multiple function calls. We did not consider this case for time reasons, and it did not appear in the evaluated game engines. In a more robust typestate checker, the typestate-changing effects of these function calls could be composed together to determine the statement's effect on the variables present in the statement.

The algorithm terminates once an entire pass is made without any changes. It is guaranteed to terminate - in case 1.), there will never be any change to that variable after the first pass, and in case 2.) the size of the set of possible typestates for that variable must always increase. At worst, the algorithm will make v*t*s passes, where v is the number of variables, s is the number of statements, and t is the cardinality of the largest set of typestates for any object in the program. Each pass takes time s, giving us a worst case running time of v*t*s*s.

Once the algorithm terminates, there is one final pass for correctness. The only case for failure is when a function is called with a pre-condition that is not guaranteed to be true: if the union of typestates of the object in predecessor statements is not the singleton set of the functiopn's precondition, then the algorithm cannot guarantee that at runtime the variable will be in the appropriate typestate to call this function. In this case the compiler exits with an error message.

The algorithm to ensure that only relevant event listeners are active at any time is very simple. Every function with post-conditions [x, y] must call either typestate_x() or typestate_y() immediately before returning. The various typestate_x functions must call removeAllEventListeners() immediately and may only add event listeners associated with typestate x. Both of these could be improved with a data flow analysis to check that typestate_x() or removeAllEventListeners() is called at some point before the function returns, instead of immediately before a return statement or after entering the function, which we did not require in our game engines. Finally, event listeners with annotated typestate x may assume that the object to which they are added is in typestate x when the function begins executing. Thus we can run a dataflow analysis similar to the one above conidering only that object's typestates, to make sure that no potentially typestate-violating functions are called when the listener is triggered.

5 Architecture

To implement typestate checking, we make three changes to the compiler.

The first is a pre-processing step as soon as abstract syntax tree has been constructed. For class definitions and function definitions we search for and remove the "typestates" and "typestate_transition" calls that the programmer uses to annotate classes with typestates and functions with pre- and post-conditions. That information is then added to the respective class definition and function definition nodes on the abstract syntax tree for later use. We also remove "typestate_listener" calls the programmer uses to annotate event listeners with typestates, and mark that function definition node as being an event listener (along with its associated typestate).

The second change is the actual typestate checking algorithm. In each function, we create a dictionary at every statement for the typestate estimates of each variable at the end of that statement. We require the predecessor and successor statements for each statement; this is straightforward to obtain from the control flow graph. The effect any statement has on the possible typestates of any variable can be found from the type of statement it is and our annotations on the class and function definition nodes of the abstract syntax tree, which is sufficient for our algorithm to run.

The third change is the event listener checking algorithm. In classes annotated with typestates, the algorithm checks all functions with post-conditions [x, y] to ensure that typestate_x() or typestate_y() function is called before returning. These typestate_x() functions that deal with event listeners must call removeAllEventListeners() immediately upon entering, and may only add event listeners that have the associated typestate.

6 Evaluation

We evaluate our success by instrumenting a prototype card game engine with typestates, whose state transition graph can be seen in Figure 1. This game is an educational game designed to teach fractions; in it, two players play monster cards and supporting power cards to make their monsters more powerful. The entire

codebase is 1000 lines of code; game engine logic takes up 200 lines. Since the game engine is controlled completely by user input, almost all of the typestate-checking work was done by the event listener typestate checking.

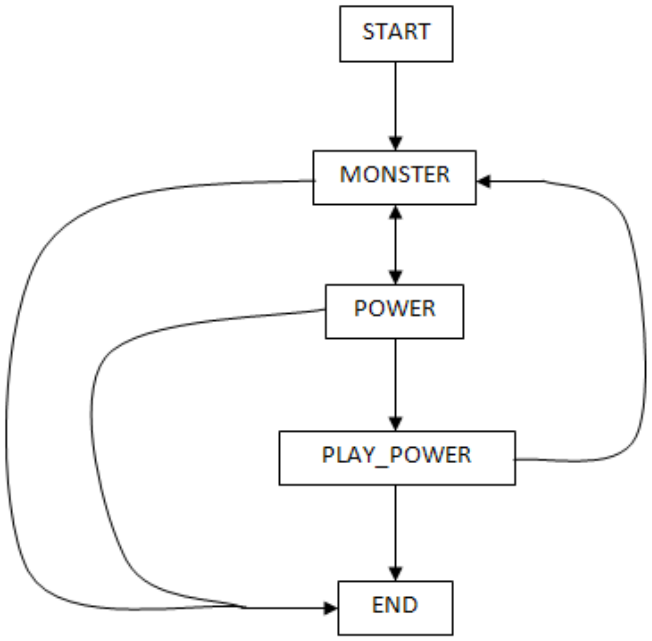

Figure 1: The possible typestates in the authors' game engine.

We also attempted to implement typestate checking on a finished game, but would not have been able to do so in a reasonable fashion. Our observations about the usefulness of our typestate checking system are as follows:

Strengths:

1. The guarantee that event listeners are only active during their associated typestate is very useful, and completely eliminates "dangling" event listeners that the programmer forgot to remove, a common source of bugs. In fact, our game engine had just such a bug which we discovered during our evaluation. The event listeners on power cards that allowed players to click them remained active even when the user was supposed to be specifying a target card for an already-clicked power card. Clicking on more power cards caused them to vanish without using their effects.

2. The system forces the programmer to clarify the set of possible game states and their associated event listeners.

3. The resulting code is easier to read and understand, as all event listeners now are labeled with their typestate and the typestate transition logic is grouped together in the typestate_x() functions.

4. The total amount of effort to implement typestate checking in our prototype was low, and mostly

consisted of creation of the typestate_x() functions and bundling together event listeners in them.

Weaknesses:

1. This typestate system does not work well in the presence of animations or other timed events. Animations all have event listeners for when they finish - so each part of each animation becomes a new typestate because it has its own event listener.

2. If listeners can be present in different combinations, this leads to a huge blowup in the number of typestates, as there must be one typestate for each combination of event listeners.

3. The system makes it difficult to create reusable UI components because each object with typestates must manage all of its event listeners. For high level objects such as game engines, this is fine. For low level objects like buttons, this is extremely undesirable - these objects must have a different set of typestates for every client, as clients will add different combinations of listeners.

4. Completely fails in the presence of aliasing. Again not a huge problem for game engines, which will not be aliased, but potentially very serious for low level objects.

5. Our algorithm does not handle arrays of objects with typestate. This is another reason typestate checking is not very useful for low level objects, as they are often stored in arrays.

Our conclusion is that our formulation of typestate checking is useful for a) high level display objects, and b) simple flash games or prototype versions of games. It seems mostly not useful for low level display objects and complex games, especially those with lots of event listeners waiting for animations or timers to finish at different times.

7 Conclusion

To the best of our knowledge, this is the first implementation of typestate checking for Actionscript 3, and the first implementation of typestate checking that takes event listeners into consideration. Typestate checking can prevent a large class of bugs related to improperly active event listeners, speeding up development and producing more stable code. Our evaluation shows that typestate checking requires more work to be usable in complex and animation-heavy systems, but is useful and may potentially speed up development for simple or prototype flash game engines with minimal alterations.

References

[1] Strom, R.E., Yemini, S.: Typestate: A programming language concept for enhancing software reliability. IEEE Trans. Softw. Eng. **12** (1986) 157–171

[2] Whaley, J., Martin, M.C., Lam, M.S.: Automatic extraction of object-oriented component interfaces. SIGSOFT Softw. Eng. Notes **27** (2002) 218–228

[3] DeLine, R., Fähndrich, M.: Typestates for objects. In: ECOOP. (2004) 465–490

[4] Lam, P., Kuncak, V., Rinard, M.: Generalized typestate checking for data structure consistency. In: Verification, Model Checking, and Abstract Interpretation. Volume 3385 of Lecture Notes in Computer Science. Springer (2005) 430–447

[5] Fink, S.J., Yahav, E., Dor, N., Ramalingam, G., Geay, E.: Effective typestate verification in the presence of aliasing. ACM Trans. Softw. Eng. Methodol. **17** (2008) 1–34

[6] Bierhoff, K., Aldrich, J.: Modular typestate checking of aliased objects. In: OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications, ACM (2007) 301–320

[7] Field, J., Goyal, D., Ramalingam, G., Yahav, E.: Typestate verification: Abstraction techniques and complexity results. Science of Computer Programming **58** (2005) 57 – 82 Special Issue on the Static Analysis Symposium 2003 - SAS'03.

[8] Aldrich, J., Sunshine, J., Saini, D., Sparks, Z.: Typestate-oriented programming. In: OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications, New York, NY, USA, ACM (2009) 1015–1022

[9] Saini, D., Sunshine, J., Aldrich, J.: A theory of typestate-oriented programming. In: Formal Techniques for Java Like Programs (FTfJP). (2010)