

The StateJava Pre-Processor

Adding States to the Java™ Programming Language

By: Peter Goodman

The StateJava Pre-Processor is a program that translates code from the StateJava programming language—a modified subset of the Java™ programming language—to valid Java™ code.

The StateJava language allows the programmer to model their classes as state machines without the hassle of having to manage states, transitions, or the details of what operations can be performed whilst an object is in a particular state.

Implementation Overview

The implementation of the pre-processor can be split into three steps: parsing, type checking, and compiling. The first and the latter are relatively straightforward and won't be extensively treated in this paper; however, the second—type checking—is more involved.

Type Checking

The first step involved in type checking is to propagate all state and method information down to every class and interface. The following sections go in-depth into the type-checking process and what it involves.

Type checking necessarily needs to be performed after all classes within the scope of the project have been parsed so that all type information necessary to check the state classes and their transitions is present.

Information Propagation

Let T be the set of all types. Let I be the set of all interfaces. Thus $I \subseteq T$. Let $K \in T$, we define $\text{Parents}(K)$ as the set of immediate supertypes of K . The following algorithm works by recursively ascending the inheritance hierarchy from some node and then propagating information back down to said node.

Propagate(K, U, F_e, F_a):

[K is the current class that we are exploring]

[U is the set of unseen types, note: U must be passed and modified by reference]

[F_e is a higher-order function to apply the propagation on two arguments: a type K and its supertype, K_p]

[F_a is a higher-order function to apply to a type K once it has received all information to be propagated]

for each $K_p \in \text{Parents}(K)$ **where** $K_p \notin I$:

if $K_p \in U$:

$U \leftarrow U - \{K_p\}$ *[by reference modification]*

Propagate(K_p, U, F_e, F_a) *[Recursively propagate information]*

$F_e(K, K_p)$ *[Propagate information from each of this type's parents]*

$F_a(K)$ *[Finish off the propagation]*

To apply the algorithm, we construct the set $U = \text{Copy}(T)$ and choose some class $K \in U$ and apply Propagate to K, U , and two higher-order functions: F_a and F_e , *until* U is the empty set.

Information propagation proceeds in two steps: the first step is to propagate the states through the inheritance hierarchy. The second step is to propagate the method transitions. In-between these two steps we construct our transition function, δ , and begin to construct our alternate transition function, γ . δ is the transition function that the pre-processor will end up compiling to Java™ code and γ is the transition function that the pre-processor will use for type-checking.

Step 1: State Propagation

Let F_e be a higher order function that takes arguments from $T \times T$ —the set of all pairs of classes, K and K_p , such that K_p is the superclass of K —and sets the states of K , $\text{States}(K) =$

$\text{States}(K) \cup \text{States}(K_p)$. We may also define a function F_a to verify certain properties of the states of a class $K \in T$ once all states have been propagated to K . For example: check that $\text{States}(K) \subseteq Q$, the set of all states.

In-Between: Transition Functions

Let Q be the set of states and Σ be the set of methods, then we define the transition function δ as $\delta: Q \times \Sigma \rightarrow Q$.

Let m and $n \in \Sigma$. We define equivalence, $m = n$, if and only if the type signatures¹, $\text{Sig}(m) = \text{Sig}(n)$ and the classes from which these methods came, $\text{Class}(m) = \text{Class}(n)$, are the same. We define method identity, $m \equiv n$, if $m = n$ and if m and n represent the same method implementation in the class file of $\text{Class}(m)$. Note, we denote the condition where $m = n$ and $\text{Not}(m \equiv n)$ as $m \cong n$. Thus Σ is the set of all methods over all class files that are identified by the equivalence relation on their type signatures and their originating classes. Let K_i and $K_j \in T$ be distinct state classes and $c_i \in \text{States}(K_i)$ and $c_j \in \text{States}(K_j)$, then $c_i = c_j$ if they are represented within their own classes by the same symbol. Thus, Q is the set of all states, and states are independent of their classes.

We define an alternate transition function, γ , as $\gamma: Q \times \text{Sig}(\Sigma) \times T \rightarrow Q$, which keeps track of all class transitions. Note: for some method $m \in \Sigma$ and some class $K \in T$, $(\cdot, \text{Sig}(m), K) \in \gamma$ if and only if $\text{Class}(m) = K$, m is in the class file of K , or $K <: \text{Class}(m)$, K is a subtype of $\text{Class}(m)$. An important aspect of γ is that it operates on method signatures as opposed to methods. γ identifies with method signatures as they uniquely identify methods within a class file; however, they overlap with previous implementations when seen in the context of inheritance.

¹ The type signature of a method is an n-tuple of its name and the names of the type-erased parameter type(s) and return type. The type signature does not include any state information. Further, if for some method $m \in \Sigma$, the name of $\text{Class}(m)$ appears in $\text{Sig}(m)$ then the class name will be replaced with the special 'Self' type. Special care needs to be taken when doing type-erasure to make sure that generic class types and inferred method types follow a consistent naming scheme.

For each class $K \in T$ and each method $m \in \Sigma$ such that $\text{Class}(m) = K$, we set $\delta(\mathbf{s}^2, m) = \mathbf{d}$ and $\gamma(\mathbf{s}, m, K) = \mathbf{d}$ for each state transition $\mathbf{s} \rightarrow \mathbf{d}$ defined on m in the class file.

By this point δ is fully defined; it contains transition information only for those methods defined in each class file but on all inherited states instead of simply the states, if any, that existed in the class file.

Step 2: Transition and Method Propagation

Let F_e be a higher order function that takes arguments from $T \times T$, K and K_p , and sets $\gamma(\mathbf{s}, \text{Sig}(\mathbf{m}), K) = \mathbf{d}$ for all $(\mathbf{s}, \text{Sig}(\mathbf{m}), K_p, \mathbf{d}) \in \gamma$ if and only if $(\mathbf{s}, \text{Sig}(\mathbf{m}), K, \mathbf{d}) \notin \gamma$. As is clear, F_e propagates all transitions from classes to their subclasses. Note: F_a is not required.

At this point γ is fully constructed and we can begin some more rigorous type checking.

Interface Checking

Interfaces are handled by requiring all methods of an interface to have a transition defined in γ on all class states of each of their implementing classes. This requirement is sensible as it then means that programmers can code to an interface and not an implementation and as a result, not have to worry whether or not a specific implementation is represented as an abstract state machine or not.

Interface checking is very simple: for each class $K \in T$ and for each interface $C_K \in I$ such that $C_K \in \text{Parents}(K)$, check that for each method $m \in \Sigma$ defined in the interface, $(\mathbf{s}, \text{Sig}(m), K, \mathbf{d}) \in \gamma$. Note: we need only do check on the immediate interfaces of some class and not seek out the interfaces that are transitively implemented by some class K as if K_p contains all transitions in γ as required by some interface $C_P \in I$ such that $C_P \in \text{Parents}(K_p)$ then K will also contain said transitions in γ .

² In this paper, unqualified variables will follow this bolded convention. Unbound variables are symbolically meaningful when used in various places.

State Reachability

The transitions on each class must be checked in order to guarantee that all states used in each class can be reached by a series of zero or more state transitions. We can use the following algorithm to determine all non-reachable states of each class:

UnreachableStates:

[Determine all of the non-reachable states of each class]

[Returns a set of tuples, $(K \in T, \{\dots\} \subseteq Q)$, the set of classes and their non-reachable states]

let $N: T \rightarrow Q^*$ *[the relation between each class and their non-reachable states]*

for each class $K \in T$:

let $P_r = \text{InitialStates}^3(K)$, $N_r = \text{Copy}(\text{States}(K))$, $J_r = \emptyset$.

[Previously reached, not reached (yet), and just reached, respectively.]

do:

$J_r = \emptyset$. *[Reset the just-reached set]*

for each state $s \in N_r$:

if $\exists d \in Q$ **such that** $(s, \cdot, K, d) \in \gamma$:

$P_r = P_r \cup \{d\}$

$J_r = J_r \cup \{d\}$

$N_r = N_r - J_r$ *[Remove any just-reached states from the not-reached set]*

while $J_r \neq \emptyset$.

$N(K) = N_r$

return N .

General Type Checking

The type checker is able to recognize and warn the programmer of the following type errors:

³ InitialStates(K) for some $K \in T$ represents the set of states to which the constructor(s) of K initialize an object of K. While it is the case that there are many initial states of a class, it is more appropriate to think of there being one and only one start state of the automata such that the start state has only those transitions defined by the constructors and performs a transition from $\cdot \rightarrow a$ for some initial state $a \in Q$.

- Missing initial state on constructor(s).
 - Base class with defined states and no initial state(s).
 - Superclass with defined states and subclass with no initial state(s).
- Missing Constructor(s).
 - Base class with defined states and no constructor(s).
 - Superclass with states/constructor(s) and subclass missing at least one of the constructor(s) of the superclass.
- Duplicate constructor signatures (independent of initial state).
- Missing methods/transitions required by interface.
- Transitions on/to states that are not part of the class's states.
- Non-deterministic/duplicate transitions, i.e. for some methods $m, n \in \mathcal{M}, \exists a, b, c \in Q$ such that $m(a \rightarrow b)$ ⁴ and $n(a \rightarrow c)$ where $m \equiv n$ and $b \neq c$ or $m \equiv n$ and a and b are not necessarily distinct. Non-deterministic transition checks operate on δ and *not* γ . γ is not used as what would appear to be a non-deterministic transition in γ during its construction that doesn't exist in δ is in fact a redirection or specialization of an inherited transition.
- State class inheriting from a non-state-class.
- Non-checkable type (warning): when information is missing from parse of the project for a particular type.
- Unreachable states.

Clearly, the above type checks operate on class information alone, i.e. the type checker does not attempt to analyze the body of methods and determine when a method call is illegal. Instead, the type checker defers to the way the pre-processor compiles the type classes to error on non-existent transitions.

Note: Many of the above type errors/warnings can be detected during the information propagation stage or when building the δ and γ functions.

⁴ We denote a transition from state a to state b on method m as: $m(a \rightarrow b)$.

State Overloading

The Java™ programming language allows method overloading according to parameter and return types and so it is natural to allow overloading to occur for different transitions as well. As a result, it is possible and many times useful for there to exist two distinct (in terms of their implementation in the class file) methods $m, n \in \Sigma$ such that $\text{Class}(m) = \text{Class}(n)$ and $\text{Sig}(m) = \text{Sig}(n)$ but where m and n have disjoint⁵ transition sets. However, given our previous definition of method equality, δ sees both these methods as equivalent and so we end up with a natural equivalence relation defined over δ . As such, the pre-processor first groups methods into their equivalence classes and merges the bodies of the methods in the compilation phase. Special care needs to be taken when performing method merging. While $\text{Sig}(m) = \text{Sig}(n)$, there is no such that the parameter *names* of m and n are the same and so the pre-processor accounts for such discrepancies.

Transition Specialization

Transition specialization, or overriding, occurs when there exist two methods $m, n \in \Sigma$ such that $\text{Sig}(m) = \text{Sig}(n)$, $\text{Class}(m) \neq \text{Class}(n)$, $\text{Class}(m) <: \text{Class}(n)$, and $\exists (\mathbf{s}, m, \mathbf{d}), (\mathbf{s}, n, \mathbf{d}) \in \delta$. We can check when a method has been specialized on a transition when we are propagating transitions and record such specializations. Recording them is important because the parent class implementation of a method—if such a superclass and method exist—must⁶ be called if a subclass's implementation of said method is called from an invalid state.

Transition Redirection

Transition redirection is a useful tool—especially in the context of state inheritance—that allows a subclass to specialize and redirect a transition of the superclass. Let $K, K_p \in T$ be distinct classes such that $K_p \in \text{Parents}(K)$ and $\text{States}(K_p) \subseteq \text{States}(K)$. Let $a, b, c \in \text{States}(K)$ be such that $a, b \in \text{States}(K_p)$ and $c \neq b$. Let $m, n \in \Sigma$ be methods such that $\text{Sig}(m) = \text{Sig}(n)$ and $\text{Class}(m) = K_p \neq \text{Class}(n) = K$ and $m(a \rightarrow b)$ and $n(a \rightarrow c)$. Thus, n replaces m in γ for K ,

⁵ The transitions are necessarily disjoint as it would be a non-deterministic transition type error if they were not.

⁶ This is an overly strong statement as it is the case in the current implementation that calling a parent implementation must be done on a state transition error as the pre-processor does not check whether or not the transition set of the subtype method closes over the transition set of the method of the supertype.

but (obviously) not for K_p . Transition redirection is especially useful when state $c \notin \text{States}(K_p)$ as it allows the state machine of K to gracefully add and change the expressive power of the state machine of K_p instead of only being able to add transitions.

Unfortunately, transition redirection presents a problem to performing static type-checking actual computation. As a result, transition redirection is a clear candidate for further research.

The Effect of the Transition Implementation on Transition Semantics

The semantics of state transitions depends heavily on how pre-processor implements the state transitions and what restrictions are put on method calls. There are two important cases to consider when implementing the compiler: self-calls to state transitioning methods within the body of a state transitioning method and indirect self-calls to state transitioning methods within the body of a foreign method called within the body of a state transitioning method. Note that the latter is a generalization of the former.

In/Direct Self-Calls

Let $m, n \in \Sigma$ be such that $\text{Class}(m) = \text{Class}(n)$ or $\text{Class}(m) <: \text{Class}(n)$. Let $a, b, c \in Q$ be such that $m(a \rightarrow b)$ and $n(a \rightarrow d)$. Let O be an object such that $\text{Class}(O) = \text{class}(m)$. Suppose $O.m$ is called and that within the definition of m , $O.n$ is called. When $O.m$ returns, the state of O will be transitioned from $a \rightarrow b$.

Suppose that the pre-processor compiled code so that state transitions would always be performed. We will denote $O:s$ for some $s \in Q$ to be the current state of the object O . Method m can be called on $O:a$ and before m returns, we call method n . Method n returns first and transitions O from $a \rightarrow c$. We may follow this with an arbitrary amount of computation, and then m returns, resulting in the transition $a \rightarrow b$ on $O:c$. Clearly, this is erroneous behavior.

Suppose instead that once we are within the scope of a method transition, i.e. once within the body of some method $O.m$ then any computation that is executed from within m will not result in the state of O changing until (the initial call to) m returns. At first this approach appears sensible; it solves the previous problem with direct calls to transitioning methods

from within the scope of another transition. However, this method suffers a more serious flaw, especially when seen in terms of indirect calls to transitioning methods.

Let O, m, n, a, b, c be as above and let $p \in \Sigma$ be a method such that $\text{Class}(p)$ cannot be related by any type equivalence to $\text{Class}(O)$. Let P be an object such that $\text{Class}(P) = \text{Class}(p)$.

Suppose that $O.m$ is called and within the scope of m , $P.p$ is called, and within the scope of p , $O.n$ is called. Suppose we narrow our focus and look only at the computation started by the invocation of $P.p$. We see that if p is called within the scope of $O.m$ then p will be able to execute arbitrary computation on O ; however, if p is not called within the scope of $O.m$ then p will be restricted to performing deterministic computation on O as defined by γ .

Clearly, the above demonstrates undefined behavior. If in/direct transition method calls can be performed then all computation performed within the scope of the initial method call cannot be type-checked. As such, the pre-processor simply disallows the in/direct transitions on a given object once in the scope of a transition on said object.

Exceptions

Currently the pre-processor compiles the StateJava classes in such a way as to severely impair Java's ability to use exceptions as a means of raising and handling errors.

Suppose that the pre-processor allowed exceptions. Let $m \in \Sigma$ be some method that acts on an object O and performs a state transition on O . If an exception is thrown within the computation in the scope of m and is not also caught within the scope of that computation then the exception will bubble up Java's call stack and impairing m 's ability to transition O into the proper state.

Note: it is the case that the pre-processor performs the state transitions within a `finally` block in the method and so would be performed regardless of exceptions. However, semantically, if a method does not complete its intended computation before the transition occurs then the object must be considered to be in an undefined state.

Thus, in order to disallow objects from entering into an invalid state, the pre-processor inhibits Java's ability to bubble exceptions up the call stack.

Non-Determinism

Suppose that the pre-processor allowed for non-deterministic transitions (as seen before). An object with non-deterministic transitions would then simulate an NFA. Let $m, n \in \Sigma$ be such that $m = n$ and $m \equiv n$ then for states $a, b, c \in Q$, $m(a \rightarrow b)$ and $n(a \rightarrow c)$. From a practical perspective, the pre-processor could compile these methods and arbitrarily choose to call the implementation of m before n ; however, what if both m and n return values? In sequential computation, one return value would need to be arbitrarily chosen unless Java allowed a method to return many values⁷ or supported an alternate form of computation⁸. Thus, this form of non-determinism is more suited toward parallel computation. The pre-processor does not currently support any form of concurrent programming constructs.

Final Thoughts

Clearly, the current implementation of the pre-processor uses a conservative approach to type-checking transitions and classes. Further work would need to be put into performing static and runtime state/type checking of actual computation so that a program doesn't simply die when an invalid transition is attempted. Also, more inspection into the type correctness of transition redirection, parallel computation, abstract classes (not supported yet), and exceptions should be done. Finally, more of the Java™ programming language should be supported, such as inner classes and anonymous classes.

⁷ The Common Lisp programming language allows a function to return many values using the `(values ...)` function/macro.

⁸ Such as continuations.