

Hanoi: A Tpestate DSL for Java

Iain McGinniss and Simon Gay

School of Computing Science
University of Glasgow
Scotland
`{iainmcg,simon}@dcs.gla.ac.uk`

Abstract. Many APIs contain interfaces in which methods have strict state based preconditions, while mainstream programming languages provide no facility to express these constraints in a concise, human readable, machine verifiable form. Failure to detect violations of these preconditions can result in partial or catastrophic failure of a system, therefore programmers must take great care in understanding and verifying the usage of such APIs. We present a domain specific language, Hanoi, which can be used to express state restrictions in APIs. By classifying Hanoi and existing tpestate languages and highlighting their differences, we show Hanoi can be more concise and potentially easier to understand than existing approaches. We also present a simple dynamic checking system which uses Hanoi models to verify interface usage, alleviating the burden on the implementer of such interfaces to defend against incorrect usage.

Keywords: tpestate, state modelling, programming language design, domain specific languages

1 Introduction

Stateful behaviour and the enforcement of state based preconditions on certain operations is crucial within a number of problem domains in software engineering, such as implementing protocols, user interfaces and data structures. The simplest and most common example of this in many languages is the concept of an iterator: an interface which abstracts away the details of linearly scanning through the contents of a data structure. Such an interface may contain the following methods:

- `hasNext()`, which indicates whether or not more items exist.
- `next()`, which returns the next available item. This must not be called if there are no remaining items (i.e. `hasNext()` returns `false`).
- `remove()`, which removes the last item returned by `next()` from the underlying data structure. May only be called once for each call to `next()`.

The iterator interface is often documented in this way — informally, using natural language and therefore not appropriate for machine verification. The

restrictions on method calls on an interface like Java’s `Iterator` can be expressed as a finite state machine. The transitions in the state machine represent method calls, and may also be conditional on the return value — one could view a call to `hasNext()` as conditionally triggering a transition to a state that allows a call to `next()` if the return value is true.

There are a number of widely studied implementation techniques [9] for implementing stateful behaviour in object oriented languages. However, regardless of which approach is taken, tools rarely exist to help implement such patterns correctly. Code must be written to store and update the current state of the object conditionally. Preconditions must be manually checked and enforced. Documentation of state specific behaviour is informal and not guaranteed to match what has actually been implemented. In essence, correctly applying a software pattern to represent and monitor stateful behaviour is often tedious and obfuscates the important implementation details of the object. Additionally, incorrect usage of the object can only be detected at runtime — very little information to aid a static analysis can be reliably inferred from the code.

Devising a simple, expressive, formalised means of describing state based preconditions has numerous benefits. The specification of state based preconditions can occur at the same time as class and interface design, and expressed in a form that is both readable to API users and by the compiler and runtime system, allowing for both static and dynamic analysis. If the compiler and runtime system take responsibility for enforcing the defined model, then implementations can be much clearer and more concise, having been freed from the task of also protecting itself from invalid usage.

The integration of finite state machines into types to represent state based preconditions is known as “typestate” [15]. In the literature, research projects either define a new language with typestates forming some integral part of the grammar (`Sing#` [13], `Spec#` [4] and `Plaid` [1], for example), or use an annotation based approach where the state machine is defined either in fragments throughout the existing code (`Plural` [5], `Fugue` [8], `ESC/Java` [7]) or as one unit (`Bica` [10]). Neither approach is ideal for the practical software engineer: annotation based definitions can be used in contemporary languages without any change to existing tools or the language, but hinder clear expression due to the syntax constraints of annotations and the fragmented nature of the definitions. It is often not straightforward to annotate existing third-party types, for instance those in the Java SDK. New or modified languages can be difficult to migrate to, for political and technical reasons, and may not be able to interact easily with existing code. In this paper, we explore the possibility of a middle ground that can be used to verify pre-existing Java code without modification: the Hanoi language. We compare Hanoi with existing approaches in more depth in Section 4.

It is worth noting that at this stage we do not implement a static analysis for Hanoi — the language is designed to make static analysis possible, but at present our implementation provides a dynamic checker only (see Section 5).

Implementing a static analysis tool for the Hanoi language is the focus of our future research, as outlined in Section 6.

1.1 Design goals

There were a number of considerations that influenced the design of Hanoi, our definition language for typestate. Definitions should, first and foremost, be easy to read and understand as a single unit. Definitions should also be easy to create and modify, so as to encourage the usage of the system and iterative refinement of models. Writing the definitions should not require any special tools — preferably, they should be plain text with a simple grammar. Using the system should not require any changes to the Java language and it should be possible to retroactively define models for existing code, such as the Java SE APIs. Finally, it should be practical to use models defined in the language to statically check that client code is safe.

The overall impact of these goals was to produce a small domain specific language for the expression of state dependent restrictions, using a type of hierarchical state machine as the underlying formalism which is inspired by Harel’s statecharts [11].

1.2 Contributions

Our work makes the following new contributions. We attempt to find a middle ground between the definition of a new programming language with typestate embedded and annotation based definition of state based preconditions on methods. We use a hierarchical state machine model which is designed for readability and ease of understanding. The model has useful properties that make iterative development and maintenance possible, and the hierarchical structure and semantics is a good match for abstract interpretation and data flow analysis. The model also allows for the definition of state transitions that are dependent upon return values and thrown exceptions, allowing for more detailed modelling than possible in other existing approaches.

2 An Introduction to Hanoi

Hanoi has a simple syntax which allows for the expression of complex state dependent behaviour. It is easier to introduce its syntax by example than by formal specification.

The Hanoi model for the Java Iterator interface is provided in Listing 1.1. Hanoi models are hierarchical finite state machines: states can have children, and all states have a common ancestor (the “top” or “root” state), which in the Iterator model is named **ACTIVE**. Each state declares a set of legal method calls in that state, which can trigger a state transition. The **ACTIVE** state declares the operations which are always legal on an **Iterator** object, as child states inherit the transitions of their parent subject to the rules described in Section 3.

Listing 1.1. Hanoi model for java.util.Iterator

```
1 ACTIVE {
2     NEXT_AVAILABLE {
3         CAN_REMOVE_MIDDLE { remove() -> NEXT_AVAILABLE }
4         next() -> CAN_REMOVE
5     }
6     CAN_REMOVE {
7         remove() -> ACTIVE
8         hasNext() :: true -> CAN_REMOVE_MIDDLE
9     }
10    hasNext() :: true -> NEXT_AVAILABLE
11    hasNext() :: <other> -> <self>
12 }
```

In the iterator example, the transition on line 10 states “hasNext() can be called, and if it returns the value `true` then the iterator is now in state `NEXT_AVAILABLE`.” Similarly, the transition within `NEXT_AVAILABLE` on line 4 states that “`next()` can be called, and the iterator will then be in state `CAN_REMOVE`” (regardless of return value, in this case). The `::` token can be read as ‘returns’, and `->` can be read as ‘transition to’.

Line 11 demonstrates two special tokens, `<other>` and `<self>`. The condition type `<other>` states that if all other conditions specified for the method call do not match the returned value, then this condition will match and the associated transition will take place. This is similar to the “default” branch on switches in Java.

A transition to `<self>` means that no transition will take place, that the object will still be in the same state. On line 11, the transition can be read as “if `hasNext()` returns a value which has not been matched, then do not cause a transition”. There is a subtle difference between this and `hasNext() :: <other> -> ACTIVE` — if one were to call `hasNext()` in the `CAN_REMOVE` state and it were to return `false`, the definition on line 11 would leave the object in state `CAN_REMOVE` while the alternative will trigger a transition to `ACTIVE`, which is undesirable as it prevents a legal call to `remove()`.

In Hanoi, the convention is that if no condition is specified then we assume the condition is `<other>`. The transition, however, must always be specified. For more information on supported condition types, see Section 3.

In addition to inheriting the transitions of a parent state, a child may also override these transitions subject to some restrictions. An example of overriding is shown on line 8, where the transition for `hasNext()` defined on line 10 is changed such that the result state will be `CAN_REMOVE_MIDDLE` instead of `NEXT_AVAILABLE`. This override exists to ensure that we do not lose the ability to call `remove()` if a call to `hasNext()` is made, for instance to ensure the simple method shown in Listing 1.2 is legal.

Listing 1.2. Method which removes the middle elements of a collection

```
1 void removeCenter(Iterator it) {
2     boolean first = true;
3     while(it.hasNext()) {
4         it.next();
5         if(!first && it.hasNext()) it.remove();
6         first = false;
7     }
8 }
```

Listing 1.3 demonstrates some additional aspects of the Hanoi grammar. Where the type being modelled is a class rather than an interface, constructor transitions are specified that indicate the initial state of the object, as shown on lines 1 and 2. As constructors are fundamentally different to methods, in that calling a constructor creates an object rather than mutates an existing object, it was decided that constructor transitions should not appear in the bodies of the state declarations. Instead, they exist outside the state hierarchy, referring to it through a “where” clause. A constructor transition must be specified for each public constructor declared on a class. As shown in the model, it often makes sense for constructors to specify an initial state that is not the root state — here, the root state corresponds to the set of methods available when the socket has reached the end of its usable life, with no transitions out of this state. When a model is structured like this, at least one constructor must necessarily place the object in a more specific child state for it to be usable.

This model also illustrates *exception transitions*, shown on lines 13 and 16. When a method has a checked exception in its signature, the model must include a transition for when this exception is thrown. In this case, the model indicates (through the slightly modified syntax of `!!`, read as “throws”) that when an `IOException` is thrown that the socket has been disconnected.

2.1 Hanoi Annotations

While the Hanoi language as described above is the primary means of providing a typestate model in our system, we also support an annotation based approach similar to that found in Fugue. The annotation equivalent to the iterator model shown in Listing 1.1 is shown in Listing 1.4. This alternative means of specification was implemented in order to illustrate that the underlying semantics of the Hanoi typestate model are independent of the mode of expression. More importantly, it also makes it possible to perform experiments in the future to directly compare the psychological aspects of working with a DSL versus an annotation based approach. See Section 6 for our plans on future work.

3 Semantics

A state must be substitutable for its parent state in Hanoi, which is guaranteed by the following rules:

Listing 1.3. A model for a simple TCP socket

```
1 new(InetSocketAddress) -> CONNECTED
2 new() -> DISCONNECTED
3 where
4 CLOSED {
5   DISCONNECTED {
6     connect(InetSocketAddress) :: true -> READABLE
7     connect(InetSocketAddress) :: false -> <self>
8   }
9   CONNECTED {
10    read() :: -1          -> <self>
11    read() :: <other>    -> <self>
12    read() !! IOException -> DISCONNECTED
13
14    write(int)          -> <self>
15    write(int) !! IOException -> DISCONNECTED
16
17    disconnect() -> DISCONNECTED
18  }
19  close() -> CLOSED
20 }
```

Listing 1.4. Part of the iterator model expressed using Hanoi annotations

```
1 @States({
2   @State(name="ALIVE"),
3   @State(name="NEXT_AVAIL", parent="ALIVE"),
4   @State(name="CAN_REMOVE_MIDDLE", parent="NEXT_AVAIL"),
5   @State(name="CAN_REMOVE", parent="ALIVE")
6 })
7 public interface Iterator {
8
9   @Transitions({
10    @Transition(from="CAN_REMOVE", to="CAN_REMOVE_MIDDLE",
11               whenResult="true"),
12    @Transition(from="ALIVE", to="NEXT_AVAIL",
13               whenResult="true"),
14    @Transition(from="ALIVE", to="<self>",
15               whenResult="false")
16  })
17   public boolean hasNext();
18
19   // ...
20 }
```

- A child state must allow at least the same set of methods as the parent, and may allow additional methods.
- If a transition overrides one or more parent transitions, the target state must be a substate of all the target states on the overridden transitions.

The conditions specified on Hanoi models can be easily translated into either numeric intervals (i.e. “[0,10]”) or finite sets of values from the return type of the method. If the return type is numeric, i.e. one of `byte`, `char`, `int`, `long`, `float`, `double`, and the boxed object types thereof, the legal set of condition operators are `<`, `<=`, `=`, `>` and `>=`. If the return type is a boolean, an enumeration or another object type, the only legal condition operator is `=`. For object types, the special value `null` may be matched against. The default operator is `=`, allowing `“x() :: 0 -> Y”` as a more convenient form of `“x() :: =0 -> Y”`.

When a set of transitions is defined for a method, all possible return values must be handled by those transitions. This also extends to the declared exceptions on a method — if a method throws an `IOException`, a transition must be declared for when this exception type is thrown. The syntax for declaring exception transitions is slightly different: `“x() !! IOException -> Y”` declares that if an `IOException` (or subtype thereof) is thrown by `x()` then the object will be in state `Y`. We do not, however require that exception transitions be declared for unchecked exception types (those which are subtypes of `RuntimeException` or `Error` in Java) — these are automatically treated as transitions to the root state. This design decision was made based upon the premise that if an unchecked exception is thrown from an object then either a programmer error has occurred, or some serious or fatal system error has occurred that has likely not been handled within the object, therefore it is likely not in a consistent state anymore. The top state often corresponds to the weakest invariant on the object, and as such is the safest choice for the target state. Such default transitions can of course be overridden like any other transition, if the implementation has in fact handled such an unchecked exception to ensure it is in a consistent state before rethrowing it.

Declared transitions cannot ‘overlap’, meaning that the set of return values matched by one transition must be disjoint from the set of return values matched by any other transition, unless one transition matches values which are all subtypes of the values matched by the other transition. These rules have some important consequences. Listing 1.5 is illegal for a number of reasons:

- The transitions defined for method `m()` overlap on the value 0. If method `m()` were to return 0, it would be unclear whether the object should be in state `X` or state `Y`.
- The transition set for method `n()` is incomplete in state `Y`. If `n` returns an integer, it would be unclear what state the object is in if the method returns a value $v \leq 0$. While the documentation may indicate that method `n()` will never return such a value, Hanoi has no way of determining this through inspecting the return type of the method. If the Java language allowed the definition of scalar subtypes, the method could be formally specified as re-

Listing 1.5. An Illegal Hanoi model

```
1 ROOT {
2   X { a() -> Y }
3   Y { n() :: >0 -> X }
4   a() -> X
5   m() :: >= 0 -> X
6   m() :: <= 0 -> Y
7 }
```

turning values in a specific range and the Hanoi model could exploit this additional information.

- The transition defined for method `a()` on line 2 conflicts with the parent definition on line 4: the target state `Y` is not a substate of state `X`. This, in turn, means that state `X` could not be safely substituted for the `ROOT` state, as the successor states after a call to `a()` offer inconsistent sets of methods.

Listing 1.6 demonstrates some more complex legal instances of transition overriding:

- The transition override of method `a()` on line 4 is legal, as state `Y` is a substate of state `X` (as targeted on line 10).
- The transition override of method `b()` on line 3 is legal, as state `X` is a substate of the `ROOT` state (as targeted on line 5).
- The transition override of method `m()` on line 6 is legal. The parent definition on line 10 has a transition to `self`, which is state `X` within the context of the definition on line 12. State `Y` is a substate of state `X`, therefore the definition is legal.
- The transition override of method `c()` on line 7 is legal. This transition overrides both of the transitions defined in the parent. The target state, `Y`, is a substate of both `X` and `ROOT`, therefore the override is legal.
- The exception transitions for method `c()` on lines 14 and 15 are legal, as `EOFException` is a subtype of `IOException` and the target state for when an `EOFException` is thrown is a substate of the target state for when an `IOException` is thrown.
- The exception transition for method `c()` declared on line 8 is legal as the target state `Y` is a substate of all the target states for thrown exceptions of type `Throwable` declared explicitly and implicitly in the parent state. `Throwable` is the supertype of all exception types, therefore the target state `Y` must be a substate of `ROOT`, which is the target state for thrown unchecked exceptions (implicitly) and `IOException`. `Y` must also be a substate of `<self>` which is the target state for when `EOFException` is thrown. In this context, `<self>` is `X`.

Great care is taken here to ensure that substitutability is possible, as this helps in making the model easier to understand and aids in static analysis: trees have

Listing 1.6. A Hanoi model with legal transition overriding

```
1  ROOT {
2    X {
3      Y { b() -> X }
4      a() -> Y
5      b() -> ROOT
6      m() :: >0 -> Y
7      c() :: <=0 -> Y
8      c() !! Throwable -> Y
9    }
10   a() -> X
11   m() -> <self>
12   c() :: <0 -> X
13   c() :: >= 0 -> ROOT
14   c() !! IOException -> ROOT
15   c() !! EOFException -> <self>
16 }
```

a natural “join” operator, so when performing a data flow analysis there is a natural and safe way of merging the state of an object when two paths through a program converge.

3.1 Formal semantics

In order to formalise the semantics of Hanoi, let us first present a formalism for a finite state machine and extend this to take a substate relation into account. A flat finite state machine for a type τ is defined as the tuple $\langle C, M, S, \alpha, \delta \rangle$ composed of the set of constructors C , method calls values M and states S along with two functions $\alpha : C \rightarrow S$ and $\delta : S \times M \times V \rightarrow S$ which define the state transitions in response to calls to constructors and methods, where for simplicity V denotes the set of all possible return values including thrown exceptions. α is a total function, i.e. there is a defined initial state for every declared constructor for the associated type τ , while δ is a partial function as there will be methods we wish to have no defined target state for certain source states. We shall refer to all such tuples that define finite state machines for a type τ as FSM_τ .

FSM_τ corresponds to a simplified form of Hanoi without inheritance, where there is no substate relation between states. The transition function δ is easily constructed directly from a transition defined in the Hanoi DSL. For instance, a transition “ $m() :: >0 -> Y$ ” defined in a state X becomes $\forall v > 0, \delta(X, m, v) = Y$.

The flat finite state machine definitions can be augmented to produce a hierarchical finite state machine $\langle C, M, S, P, \alpha, \beta \rangle$ which includes two new components: a relation $P \subset S \times S$ which defines the parent for each state, and a

partial function $\beta : S \times S \times M \times V \rightarrow S$ defining a substate-aware variant of δ . We shall refer to all such tuples that define hierarchical finite state machine for a type τ as HFSM_τ .

The state inheritance relation P is defined such that:

- The top state ($\top \in S$) has no parent: $\nexists s' \in S \text{ s.t. } (s', \top) \in P$
- Each non-root state has exactly one parent:
 $\forall s \in S/\{\top\}, \exists! s_p \in S \text{ where } (s_p, s) \in P$

The full substate relation $<:$ is the reflexive, transitive closure of P .

$\beta(s_{\text{src}}, s_{\text{def}}, m, v)$ defines the state transition from the “source” state $s_{\text{src}} \in S$ when method $m \in M$ is called returning value $v \in V$. s_{def} is the “defining state” (which is a superstate of s_{src} , i.e. $s_{\text{src}} <: s_{\text{def}}$) where the applicable transition is actually declared. It is necessary to distinguish s_{src} and s_{def} in order to correctly handle transitions to **<self>**: a transition “ $\mathbf{x}() \rightarrow \mathbf{<self>}$ ” declared on a state X is equivalent to $\forall s_{\text{src}} <: X, v \in V \beta(s_{\text{src}}, X, x, v) = s_{\text{src}}$, while a transition to a specific state such as in “ $\mathbf{x}() \rightarrow Y$ ” is equivalent to $\forall s_{\text{src}} <: X, v \in V \beta(s_{\text{src}}, X, x, v) = Y$.

For a hierarchical finite state machine to be considered a valid Hanoi state machine, it must also conform to the following additional rules:

- All return values for a legal method must be covered:
 $\langle s, s, m, v \rangle \in \text{dom}(\beta) \Rightarrow \forall v' \in V, \langle s, s, m, v' \rangle \in \text{dom}(\beta)$
- Transitions are inherited:
 $\forall s_2 <: s_1, \langle s_1, s_1, m, v \rangle \in \text{dom}(\beta) \Rightarrow \langle s_2, s_1, m, v \rangle \in \text{dom}(\beta)$
- Inherited transition targets must be covariant:
 $\forall s_3 <: s_2 <: s_1, \langle s_2, s_1, m, v \rangle \in \text{dom}(\beta) \Rightarrow \beta(s_3, s_1, m, v) <: \beta(s_2, s_1, m, v)$
- Transition overrides must be covariant:
 $\forall s_3 <: s_2 <: s_1, s' = \beta(s_3, s_2, m, v) \wedge s = \beta(s_3, s_1, m, v) \Rightarrow s' <: s$
- The source state must always be a substate of the defined state:
 $\langle s_{\text{src}}, s_{\text{def}}, m, v \rangle \in \text{dom}(\beta) \Rightarrow s_{\text{src}} <: s_{\text{def}}$.

From β we can construct a recursive function γ which takes transition overriding into account:

```

 $\gamma(s_{\text{src}}, s_{\text{def}}, m, v) =$ 
  if  $\langle s_{\text{src}}, s_{\text{def}}, m, v \rangle \in \text{dom}(\beta)$ 
  then  $\beta(s_{\text{src}}, s_{\text{def}}, m, v)$ 
  else  $\gamma(s_{\text{src}}, s_p, m, v)$  when  $\exists (s_p, s_{\text{def}}) \in P$ 

```

$\gamma(s_{\text{src}}, s_{\text{def}}, m, v)$ is therefore only defined when $\beta(s_{\text{src}}, s, m, v)$ is defined for an s such that $s_{\text{def}} <: s$.

It is worth noting that these definitions do not take into account exception transitions, where a return value v has an associated type and corresponding type hierarchy that influences the choice of the successor state. This has been omitted for simplicity — it requires a further modification to β and γ to take into account this type and find the most specific transition for the value type, with the restriction that the target state is covariant with the value type.

3.2 Proof of safe state substitutability

From these definitions, we can prove that a substate can be safely substituted for its parent state. This informally means that whenever we have a value v of type τ in state s_2 (written $v : \tau[s_2]$) we can use v in wherever a value of of type $\tau[s_1]$ is required as long as $s_2 <: s_1$. We require some additional definitions to prove this statement.

Definition 1 (Interaction Traces) *Let $I \subseteq M \times V$ be the set of possible method calls and their associated possible return values for a type, which we will refer to as interactions. Then the set of interaction traces of $\tau[s]$, referred to as $Tr(\tau[s])$, is defined inductively as:*

$$\frac{}{\epsilon \in Tr(\tau[s])} \quad \frac{\gamma(s, s, m, v) = s' \quad t \in Tr(\tau[s'])}{(m, v)t \in Tr(\tau[s])}$$

Lemma 1 (γ is covariant) *Suppose $s_2 <: s_1$. If $\gamma(s_1, s_1, m, v)$ is defined then $\gamma(s_2, s_2, m, v)$ is also defined such that $\gamma(s_2, s_2, m, v) <: \gamma(s_1, s_1, m, v)$.*

Proof. Let $s_0 \in S$ with $s_1 <: s_0$, with $\langle s_0, s_0, m, v \rangle \in dom(\beta)$, and with no $s_{mid} \in S$ such that $s_{mid} \neq s_0$ and $s_1 <: s_{mid} <: s_0$ with $\langle s_{mid}, s_{mid}, m, v \rangle \in dom(\beta)$. Then it follows that $\gamma(s_1, s_1, m, v) = \beta(s_1, s_0, m, v)$, which by the definition of β must be defined with $\beta(s_1, s_0, m, v) <: \beta(s_0, s_0, m, v)$. We must now show that in all cases $\gamma(s_2, s_2, m, v) <: \beta(s_1, s_0, m, v)$. There are two cases to consider:

- There are no overrides defined for the method-value pair (m, v) in the inheritance chain between s_2 and s_1 . In this case, $\gamma(s_2, s_2, m, v) = \beta(s_2, s_0, m, v)$ which, by the definition of β , must be defined and is a substate of $\beta(s_1, s_0, m, v)$.
- There exists an s_{mid} such that $s_2 <: s_{mid} <: s_1$ (where s_2 and s_{mid} may be equal), with $\langle s_{mid}, s_{mid}, m, v \rangle \in dom(\beta)$. In this case, we have that $\gamma(s_2, s_2, m, v) = \beta(s_2, s_{mid}, m, v)$ which by the definition of β must be defined and must be such that $\beta(s_2, s_{mid}, m, v) <: \beta(s_2, s_0, m, v)$, and $\beta(s_2, s_0, m, v) <: \beta(s_1, s_0, m, v)$.

In both cases, we have $\gamma(s_2, s_2, m, v) <: \gamma(s_1, s_1, m, v)$. □

Theorem 1 (Trace inclusion) *Suppose $s_2 <: s_1$. Then $Tr(\tau[s_1]) \subseteq Tr(\tau[s_2])$.*

Proof. We shall prove by induction that $t \in Tr(\tau[s_1]) \Rightarrow t \in Tr(\tau[s_2])$.

Base case: $t = \epsilon$. By the definition of Tr , $\epsilon \in Tr(\tau[s_2])$.

Induction step: Let us assume that $\forall s' <: s, t' \in Tr(\tau[s]) \Rightarrow t' \in Tr(\tau[s'])$. Suppose $t = (m, v)t'$, and that $t \in Tr(\tau[s_1])$. For t to be in $Tr(\tau[s_1])$ there must be an $s_3 = \gamma(s_1, s_1, m, v)$, with $t' \in Tr(\tau[s_3])$. As $s_2 <: s_1$, there also exists an $s_4 = \gamma(s_2, s_2, m, v)$. By Lemma 1, $s_4 <: s_3$. It follows by our induction hypothesis that $t' \in Tr(\tau[s_4])$. Therefore, $(m, v)t' \in Tr(\tau[s_2])$. □

This theorem demonstrates that in all cases we can safely use a value of type $\tau[s_2]$ wherever we can use a value of type $\tau[s_1]$, as all sequences of method calls on the latter are also possible on the former.

3.3 Model Subtyping

In addition to the substate relationship between parent and child states within a particular model, Hanoi also defines an extended form of safe substitutability between values of different base types.

Definition 2 (Behavioural subtyping) *A typestate $\tau_2[s_2]$ is considered to be a behavioural subtype of $\tau_1[s_1]$, written $\tau_2[s_2] \prec: \tau_1[s_1]$, if the following is true:*

- τ_2 is a subtype of τ_1 .
- The set of interaction traces of $\tau_1[s_1]$ is a subset of the set of interaction traces of $\tau_2[s_2]$: $Tr(\tau_1[s_1]) \subseteq Tr(\tau_2[s_2])$.

The degenerate case of behavioural subtyping is where we have two types $\tau_2 \prec: \tau_1$ which are both stateless, i.e. there are no state based preconditions on their respective methods. This is equivalent to both types have a top state defined which allows all method calls with a transition to “<self>”. While τ_2 may define additional methods, we still have that $\tau_2[\top] \prec: \tau_1[\top]$ as τ_2 must support all methods of τ_1 by the normal definition of subtyping.

In the case where the types both have Hanoi models, the standard substitutability rules in Java are no longer sufficient for our needs. Consider a simple transactional data structure such as `TransactionalMap` shown in Listing 1.7. The interface `Map` allows any ordering of calls on its methods, while `TransactionalMap` has restrictions as defined in Listing 1.8: a transaction must be started before the map can be manipulated. Our intuition here should be that `TransactionalMap[TRANSACTION] \prec: Map[\top]`, which is true as the state `TRANSACTION` allows all possible sequences of calls allowed by `\top` in `Map`. However, neither `ACTIVE` or `NO_TRANSACTION` satisfy this property. This reflects the intuition that if we have a method that expects a `Map` as a parameter, we can only safely give it a `TransactionalMap` if it is in state `TRANSACTION`, as otherwise an illegal method may be invoked.

This has important implications for type systems: method signatures must be able to indicate not only the required types of their parameters and return values, but also the required states.

Behavioural subtyping can be verified by building a simulation relation between the states of two types. The only restriction we impose for a Hanoi model of a subclass to be valid relative to the superclass is that the simulation relation should not be empty. We do not require that $(\tau_1[\top], \tau_2[\top])$ be in the relation as this is overly restrictive, as shown in the `TransactionalMap` example.

4 Classifying and Comparing Typestate Systems

Languages for expressing typestate constraints can be categorised based upon design choices over three traits:

- States can be either explicit or implicit.

Listing 1.7. A Map interface with a transaction subtype

```
1 public class TransactionalMap<K,V>
2     implements Transaction, Map<K,V> {
3     public TransactionalMap() { /*...*/ }
4     // methods from Transaction:
5     public void start() { /*...*/ }
6     public void commit() { /*...*/ }
7     public void rollback() { /*...*/ }
8     // methods from Map
9     public V get(K key) { /*...*/ }
10    public V put(K key) { /*...*/ }
11    public void remove(K key) { /*...*/ }
12 }
```

Listing 1.8. Hanoi model for TransactionalMap type in Listing 1.7

```
1 new() -> NO_TRANSACTION
2 where
3 ACTIVE {
4     NO_TRANSACTION {
5         start() -> TRANSACTION
6     }
7     TRANSACTION {
8         get(K) -> <self>
9         put(K) -> <self>
10        remove(K) -> <self>
11        commit() -> NO_TRANSACTION
12        rollback() -> NO_TRANSACTION
13    }
14 }
```

Listing 1.9. Plural grouping of transitions by method

```
1 @Cases({
2   @Perm(requires="unique(this!fr) in CR",
3     ensures="result == false " +
4       "> unique(this!fr) in CR"),
5   @Perm(requires="unique(this!fr) in CR",
6     ensures="result == true " +
7       "> unique(this!fr) in CRM"),
8   /* ... */
9   @Perm(requires="unique(this!fr)",
10     ensures="result == false " +
11       "> unique(this!fr)"),
12   @Perm(requires="unique(this!fr)",
13     ensures="result == true " +
14       "> unique(this!fr) in NA")
15 })
16 boolean hasNext();
```

- The language can focus on legal behaviour (a positive focus) or illegal behaviour (a negative focus).
- The language can be integrated to, or separated from its host language.

In these terms Hanoi is an explicit, positive, separated language. Tracematches are an implicit, negative, separated language due to the use of regular expressions that match illegal sequences. Plural, Fugue and Bica are all explicit, positive, integrated languages.

We shall now compare Hanoi, Plural and Tracematches to illustrate the impact of these design choices.

4.1 Plural

Plural is an advanced tpestate specification system which includes a hierarchical, parallel state machine model with fractional permissions to cope with aliasing [5]. Plural defines the state space and transitions using Java annotations. The primary difference between Hanoi and Plural beyond the use of annotations is the way in which transitions are logically grouped. In Hanoi, transitions are grouped by their source state. In Plural, transitions are grouped by the method causing the transition, in a Hoare logic pre- and post-condition style (as shown in Listing 1.9).

Grouping by method makes it easier for a user to determine the set of states in which it is legal to call a method, while grouping by source state makes it easier to determine the set of methods which are legal in a particular state. Grouping by source state is more practical when writing code to use an object in a known input state, which is often the case when manipulating parameters passed to a method.

The lack of a concept of transitions to “self” in Plural means that many more transitions may need to be declared in comparison to an equivalent Hanoi model — one override for each method/state pair where there is a transition to self in Hanoi (these are omitted in Listing 1.9 for brevity).

Plural supports conditional state transitions, but only for boolean return types. This means that some transitions cannot be accurately modelled, for instance checking the `size()` of a queue is greater than 0 as an alternative to checking the `isEmpty()` method, or using an `enum` return type as a means of distinguishing multiple outcomes of a method.

Plural’s primary advantage over Hanoi as a modelling language is the ability to express alias constraints, which is important for both static analysis and communicating the expected usage pattern of an object to the user. Objects often implicitly have aliasing constraints — the vast majority of objects with mutable state expect a “single owner” strategy. If sharing is permitted it is often of a very restricted form, such as access to disjoint aspects of the object’s functionality. Plural can express such aliasing models well, while Hanoi does not yet attempt to tackle this issue due to its initial focus on dynamic checking (see Section 5).

4.2 AspectJ Tracematches

AspectJ is an aspect oriented extension to Java [14]. Tracematches are a special form of join-point that allow a piece of advice (a fragment of code) to be executed when a sequence of calls matching a regular expression is detected [2]. Tracematches, by their association to aspect oriented programming, are runtime oriented and therefore not specifically designed for static analysis; however, significant and interesting research has been undertaken to optimise their runtime behaviour [3, 6]. When naïvely implemented, tracematches are extremely detrimental to performance, and so employing static analysis techniques to detect where a tracematch will never be triggered is vital to reducing such overhead.

Tracematches support more general use cases than enforcing typestate restrictions, but they can offer a simple and effective way of detecting typestate violations. Tracematches allow the binding of free variables — a very powerful technique that allows for the monitoring of multiple related objects. For instance, in Java it is illegal to modify a `Collection` while an `Iterator` is in use on that collection: the iterator is effectively invalidated when the `Collection` is modified, and any further usage triggers an unchecked `ConcurrentModificationException`. This kind of restriction cannot be specified or detected by Plural or Hanoi, as they deal exclusively with modelling the states of single objects, while tracematches can capture the association between objects and monitor them as a group.

The most interesting difference from a modelling perspective between Tracematches and Hanoi is the use of regular expressions combined with an emphasis on illegal behaviour. Both define regular languages, but the regular language defined by a tracematch is the complement of the language defined by a Hanoi model. Depending on the behaviour being modelled, using a regular expression

Listing 1.10. Regular grammar capturing illegal sequences for an Iterator

```
1 START = CN
2 CN = hNt NA | hNf CN | n | r
3 NA = hNt NA | hNf NA | n CR | r
4 CR = hNt CRM | hNf CR | n | r CN
5 CRM = hNt CRM | hNf CRM | n CR | r NA
```

Listing 1.11. Part of the regular expression for Iterator violations

```
1 ( hnf | hnt (hnt | hnf)* n (hnf | hnt (hnt | hnf)* n
2 | hnt (hnt | hnf)* r (hnt | hnf)* n)* r
3 )*
4 ( r | n | hnt (hnt | hnf)* r
5 | hnt (hnt | hnf)* n (hnf | hnt (hnt | hnf)* n
6 | hnt (hnt | hnf)* r (hnt | hnf)*n)*
7 (n | hnt (hnt | hnf)* r (hnt | hnf)* r)
8 )
```

can produce a much more compact representation, or a significantly larger one. In the worst case, converting between a finite state machine representation of size n and a regular expression representation can increase the size of the model by more than 2^n . It can also be quite difficult to manually construct a regular expression that will match all illegal occurrences — Listing 1.11 shows such a regular expression for illegal sequences of calls on an iterator, generated from the regular grammar shown in Listing 1.10, where the state names and method names have been truncated to just their initials, with `hNf` meaning “a call to `hasNext()`, returning false”, similarly for `hNt`.

One important aspect of modelling typestate from a user’s perspective is the ability for the model to explain failures — when a typestate violation is detected, it is helpful to know why the detected sequence is illegal. By using meaningful state names in a Hanoi model, we can extract useful error messages of the form “unable to call method `x()` in state `S`”, while with a typestate language that does not have explicit states we cannot provide any context. We can restore this context to an extent by using multiple complementary regular expressions that capture particular patterns of failure, for instance with the Iterator example we can have the simple regular expression `(hnf next)` to capture the attempt to call `next` when `hasNext()` has explicitly reported that there are no more elements, or `(n r r)` for an attempt to call `remove()` twice for one call to `next`. It is difficult however to construct such a set of regular expressions and be certain, without tool support, that the union of the sequences captured covers the full set of illegal sequences.

5 Runtime Checking of Hanoi Models

Hanoi can be used at runtime, as a means of verifying that calls made to a type with a Hanoi model are legal as it is being used. There are two implementations of dynamic checking with different requirements and constraints: one based on dynamic proxies, while the other uses an AspectJ load-time weaving approach. Those interested in experimenting with Hanoi are welcome to download the current implementation from <http://bitbucket.org/iaimcgin/hanoi>

5.1 Dynamic Proxies

Java’s runtime system makes it possible to implement an interface dynamically through the use of a “dynamic proxy”, which is an object which handles method calls of a type which was unknown at compile time through the use of reflection.

This facility is leveraged in Hanoi to allow instances of objects which implement a Hanoi modelled interface to be wrapped and monitored. An already constructed instance of an object can be passed to the Hanoi API to be wrapped, or a reference to the class object may be passed along with the set of parameters that would be passed to the constructor. In the former case the current state of the object must also be indicated (or it will be assumed the current state is \top for that type), while in the latter the initial state can be deduced from the set of constructor transitions declared in the associated Hanoi model.

All calls made to the proxy are checked for legality against the currently known state of the real implementation. If the call is legal, then the real implementation is invoked and its return value checked to determine what the new state is, before returning this result to the original caller. If the call is illegal, then an `IllegalStateException` can be thrown, which is useful for strict monitoring of the object as a replacement for manual defensive programming. An alternative option is to simply log the violation and cease monitoring the object, which can be useful for debugging a system without altering its behaviour.

There are two drawbacks to using dynamic proxies in Java. Firstly, only interfaces can be proxied, which prevents monitoring the usage of concrete types like `InputStream`. Secondly, wrapping the proxy around the real implementation is not automatic — the Hanoi API for building a proxy must be used in place of the object’s original constructor. The dynamic proxy based implementation does not require any additional dependencies beyond the Hanoi runtime library, and requires no changes to the configuration of the runtime environment, making it particularly well suited to prototyping and testing.

5.2 AspectJ Load-time Weaving

In order to work around the constraints of the dynamic proxy based runtime checking implementation, a code generator was implemented that creates aspects for monitoring arbitrary types. This code generator takes a set of JAR files as input and scans them for types which have associated hanoi state models (either in DSL form or direct annotation form) and produces an aspect for each found

Listing 1.12. Using Hanoi with load-time weaving

```
1 java -javaagent:aspectjweaver.jar
2     -cp main.jar:hanoi.jar:hanoi_aspects.jar
3     com.example.Main
```

Listing 1.13. A generated piece of advice to monitor a method

```
1 boolean around(): call(public boolean isEmpty())
2     && target(BoundedQueue) {
3     Object target = thisJoinPoint.getTarget();
4     IState st = getState(target);
5     Method m = BoundedQueue.class.getMethod("isEmpty");
6
7     if(!st.isLegalCall(m)) {
8         reject(m + " not legal in state " + st);
9     }
10
11    try {
12        boolean result = proceed();
13        putState(target, st.nextState(m, result));
14        return result;
15    } catch(java.lang.RuntimeException e) {
16        putState(target, st.nextState(m, new ThrownEx(e)));
17        throw e;
18    } catch(java.lang.Error e) {
19        putState(target, st.nextState(m, new ThrownEx(e)));
20        throw e;
21    }
22 }
```

type that will intercept all calls to constructors and public methods of that type. The aspect behaves in a fundamentally similar fashion to the dynamic proxy based approach - calls are checked for legality against the currently known state and rejected if they are illegal (as shown in Listing 1.13). However, the aspect based approach can easily monitor concrete types and does not require any changes to the code in order to be activated. The generated aspects are compiled and placed into a JAR file which is placed on the classpath of the program to be monitored, and the AspectJ load-time weaving agent does all the necessary bytecode transformation work to introduce the monitoring where necessary. This small modification to the configuration of the environment, as shown explicitly in Listing 1.12, is all that is required.

6 Future Work

The focus of our research to date has been on the practical and linguistic aspects of modelling tpestate. We are now investigating the implementation of a modular static analysis for Hanoi in Java. There are a number of possibilities: A subset of Hanoi can be translated into Plural annotations, tracematches and ESC/Java specifications, all of which are good options for creating a baseline analysis in order to rate the success of our future efforts. The model itself does not contain any specific support for defining roles or aliasing constraints, which is an important aspect of a realistic tpestate model. We plan to investigate the linguistic aspects of alias constraint specification, and devise a new version of Hanoi which includes such constraints in the language. We believe inspiration can be drawn from Plural and multi party session type specification languages such as Scribble [12].

One complementary aspect of defining a stateful contract, as Hanoi does, is ensuring that the implementer also obeys the contract. We intend to investigate using a model checking approach to verifying that an implementation conforms to the tpestate specification, similar to the strategy employed in Bica [10]. Model checking also affords the interesting opportunity to infer the invariants of each state, based on the observed correlation between states and internal field states. Such invariants could be particularly useful to a maintainer of a Hanoi modelled type when making decisions as to whether adding a particular piece of code to a method is safe for all states in which that method may be called.

As the main contribution of this paper centers on the linguistic aspects of tpestate specification, we believe it is important to experimentally verify that the language improves on existing attempts. Our hypothesis is that an annotation based specification is harder to read and understand than the semantically identical description provided by the Hanoi DSL. We intend to run experiments in the near future where we will test participants on the following tasks:

- Reading and interpreting the restrictions of models, tested answering simple yes/no questions as to whether a sequence of method calls is permitted by a provided model.
- Writing models based on informal “javadoc” style documentation, tested by compiling such models and testing whether they admit and reject input sequences appropriately.

7 Conclusion

Hanoi provides a simple, expressive model for stateful interfaces and a runtime tool for Java that can be used to verify that the usage of such interfaces is correct. Models can be easily written both for existing and new interfaces. With further research into the usage of stateful interfaces when aliasing and concurrency are present, we believe that this technique will eliminate a number of common, often catastrophic, programming errors through complementary static and dynamic analysis approaches.

Acknowledgements

The first author is supported by a studentship from the Scottish Informatics and Computing Science Alliance (SICSA). The work has also been supported by the EPSRC project “Engineering Foundations of Web Services: Theories and Tool Support” (EP/E065708/1).

We would also like to thank Joseph Sventek (University of Glasgow) for his advice and suggestions that led to the creation of the runtime aspects of Hanoi, David Aspinall (University of Edinburgh) for his suggestions on the possible directions in which Hanoi could be taken in future, and Nils Gesbert (INRIA Grenoble - Rhône-Alpes) for fruitful discussions related to subtyping in Hanoi.

References

1. Aldrich, J., Sunshine, J., Saini, D., Sparks, Z.: Tpestate-oriented programming. In: OOPSLA '09. pp. 1015–1022 (2009)
2. Allan, C., Avgustinov, P., Christensen, A., et al.: Adding trace matching with free variables to AspectJ. In: OOPSLA '05. pp. 345–364 (2005)
3. Avgustinov, P., Tibble, J., de Moor, O.: Making trace monitors feasible. OOPSLA '07 pp. 589–608 (2007)
4. Barnett, M., Leino, R.M., Schulte, W.: The Spec# Programming System: An Overview. In: CASSIS '05. pp. 49–69 (2005)
5. Bierhoff, K., Beckman, N.E., Aldrich, J.: Practical API Protocol Checking with Access Permissions. In: ECOOP '09. pp. 195–219 (2009)
6. Bodden, E.: Verifying finite-state properties of large-scale programs. Ph.D. thesis, McGill University (2009)
7. Chalin, P., Kiniry, J.R., Leavens, G.T., Poll, E.: Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In: FMCO '06. pp. 342–363 (2006)
8. DeLine, R., Fähndrich, M.: The Fugue protocol checker: Is your software baroque? Tech. Rep. MSR-TR-2004-07, Microsoft Research (2004)
9. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional (1994), 978-0201633610
10. Gay, S.J., Vasconcelos, V.T., Ravara, A., et al.: Modular session types for distributed object-oriented programming. In: POPL '10. pp. 299–312 (2010)
11. Harel, D.: Statecharts: A visual formalism for complex systems. *Science of computer programming* 8(3), 231–274 (1987)
12. Honda, K., Brown, G.: Scribble (2010), <http://jboss.org/scribble>
13. Hunt, G., Larus, J., Abadi, M., et al.: An Overview of the Singularity Project. Tech. rep., Microsoft Research (2005)
14. Kiczales, G., Hilsdale, E., Hugunin, J., et al.: An Overview of AspectJ. In: ECOOP '01. pp. 327–354 (2001)
15. Strom, R.E., Yemini, S.: Tpestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering* 12(1), 157–171 (1986)