

Hanoi: A Practical Typestate Model for Java

Iain McGinniss and Simon Gay

Department of Computing Science
University of Glasgow,
Lilybank Gardens,
Scotland,
G12 8QQ
{iainmcg, simon}@dcs.gla.ac.uk
<http://dcs.gla.ac.uk/~iainmcg>

Abstract. Many modern 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 simple modelling language, Hanoi, which can be used to express state restrictions in APIs. We have found it to be useful for the runtime verification of interface usage in Java, and alleviates the burden on the implementer of such interfaces to defend against incorrect usage.

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 protocol stacks, user interfaces and data structures. The simplest and most common example of this in many modern languages is the concept of an iterator: an interface which abstracts 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 [3] for object oriented languages. However, regardless of which approach is taken, the programmer has to explicitly implement the stateful behaviour, which has a number of drawbacks:

- State based preconditions must be checked manually by the implementer. This typically involves throwing an unchecked exception, as a compromise between preventing incorrect calls and avoiding the need for try/catch blocks in correct client code.
- Boilerplate code to manage storage of the current state and state transitions upon certain operations must be written. Despite the existence of well documented patterns to do this, it is still time consuming and error prone.
- Documentation of state specific behaviour is informal and not machine verifiable. The style in which such restrictions are documented varies wildly, and gaining a clear understanding of complex interfaces can be difficult.

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 — no information to aid a static analysis can be inferred from the code.

There would be two substantial benefits to delegating responsibility for verifying correct usage of stateful interfaces to the compiler and runtime system. Firstly, the implementer would be freed to focus on implementing the core functionality of the interface correctly. Secondly, users of the interface would be notified of incorrect usage at a much earlier stage. Therefore, support for statically and dynamically checking usage of stateful interfaces is desirable.

1.1 Related work

Typestate is the tight integration of state machines into types [8]. In the literature, research projects either define a new language with typestates forming some integral part of the grammar (Sing# [6] and Plaid [1], for example), or use an annotation approach where the state machine is defined either in fragments throughout the existing code, or as one unit (Bica [4] and Plural [2]). Neither approach is ideal for the practical software engineer: annotation based definitions can be used in modern languages without any change to existing tools or the language, but are difficult to understand due to the syntax constraints of annotations and the fragmented nature of the definitions. New languages can be difficult to integrate into the developer’s tool chain, 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.

1.2 Design goals

There were a number of considerations that influenced the design of Hanoi, our definition language for typestate. These were:

- Definitions should, first and foremost, be easy to read and understand as a single unit.
- Definitions should be easy to create and modify, so as to encourage the usage of the system and iterative refinement of models.
- Definitions should not require any special tools to maintain — preferably, they should be plain text with a simple grammar.
- Usage of Hanoi should not require any changes to the Java language or core runtime.
- It should be possible to retroactively define models for existing code, such as the Java SE APIs.
- It should be practical to use the model to statically check that client code is safe.

The overall impact of these design goals was to produce a small domain specific language for the expression of state dependent restrictions, based on a subset of the capabilities of Harel’s statecharts [5].

1.3 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 fragmented definition of state based preconditions on methods. In contrast to session types as defined in Bica, we use a nested state machine model which is designed for readability and ease of understanding as its primary design consideration.

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 nested finite state machines: states can have children, and all states are a child of `DEFAULT`. Each state declares a set of legal method calls in that state, which can affect a state transition. The `DEFAULT` state essentially declares the operations which are always legal on an object. Child states inherit the transitions of their parent subject to the rules described in Section 3.

In the iterator example, the transition on line 9 states “`hasNext()` can be called, and if it returns the value `true` then the iterator is now in state `HAS_NEXT`.” Similarly, the transition within `HAS_NEXT` states that “`next()` can be called, and

Listing 1.1. Hanoi model for `java.util.Iterator`

```

1  HAS_NEXT {
2    next() -> CAN_REMOVE
3  }
4
5  CAN_REMOVE {
6    remove() -> DEFAULT
7  }
8
9  hasNext() :: true -> HAS_NEXT
10 hasNext()

```

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’. If no `->` is present, no transition will take place; that is, the iterator will be left in the current state. If no `::` is present, then the transition (if specified) will take place if all other conditions specified evaluate to false. So, line 10 effectively means “if `hasNext()` returns `false`, no state transition will take place”, as the only value which is not specified in another transition here is `false`. This can be written more explicitly in Hanoi as `hasNext() :: other -> <self>`.

As a second example, consider Listing 1.2, which defines the model for an `InputStream` in Java. The nesting structure is more complex here, and in part this is because `InputStream` is a poorly defined interface: it includes methods which child implementations *may* support. The `mark(int)` and `reset()` methods are only supported by some types of input stream, and this can be determined generically by checking the return value of `markSupported()`. However, the `skip(long)` method may also not be supported, and there is no way to determine this at runtime, indicating that this method does not belong in the `InputStream` class.

The `InputStream` model illustrates inheritance in Hanoi: if we are in the `MARKED` state, we are still able to call `read()` and `close()` as they are legal transitions in the parent states. The `close()` method can be called from any state within the model (as it is defined in `DEFAULT`), and it can be called multiple times. This reflects the documentation of the `InputStream` interface — once a stream has been closed all other methods become unavailable, and `close()` is an idempotent operation — subsequent invocations have no effect.

3 Semantics

Hanoi models have some key properties:

Listing 1.2. Hanoi model for java.io.InputStream

```

1 OPEN {
2   READABLE {
3     MARKABLE {
4       MARKED { reset() -> MARKABLE }
5       mark(int) -> MARKED
6     }
7     markSupported() :: true -> MARKABLE
8     markSupported()
9     read(byte[],int,int) :: -1 -> OPEN
10    read(byte[],int,int)
11    read(byte[]) :: -1 -> OPEN
12    read(byte[])
13    read() :: -1 -> OPEN
14    read()
15    skip(long)
16  }
17  available() :: >0 -> READABLE
18  available()
19 }
20 close() -> DEFAULT

```

- Child states are subtypes of their parent states. That is, the rules defined in child states must be substitutable for those in parent states, without a logical contradiction arising.
- Transitions must be ‘complete’, meaning that if a set of transitions are declared for a method, that set must cover all possible return values of the method.
- Transitions cannot ‘overlap’, meaning that conditions specified on them must be disjoint from all other conditions related to that method.
- An implementation of an interface can only ever be in one state at a time.

These properties have some important consequences. Listing 1.3 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 has moved to 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 between `Integer.MIN_VALUE` and 0. While the documentation may indicate that method `n()` will never return a value less than 1, Hanoi has no way of determining this through inspecting the return type of the method.
- The transition defined for method `a()` on line 1 conflicts with the parent definition on line 3: 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 DEFAULT

Listing 1.3. An Illegal Hanoi model

```

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

```

Listing 1.4. A Hanoi model with legal transition overriding

```

1 X {
2   Y { b() -> X }
3   a() -> Y
4   b() -> DEFAULT
5   m() :: >0 -> Y
6 }
7 a() -> X
8 m()

```

state, as they disagree on what state the object should be in after a call to `a()`.

We call the final violation a “transition narrowing” violation in our model. This is due to its similarity to return type narrowing in object oriented languages: an override of a method in a child class can return a more specific type than its parent, but not a more general one. For example, if a method in the parent interface were to return a `List`, then it would be legal for a child interface to override this method declaration and return a `LinkedList`, but not a `Set`, as `Set` is not a subtype of `List`.

Transition narrowing is allowed in Hanoi, and is shown in Listing 1.4:

- The transition override of method `a()` is legal on line 3 as state `Y` is a substate of state `X` (as targeted on line 7).
- The transition override of method `b()` is legal on line 2 as state `X` is a substate of the default state (as targeted on line 4).
- The transition override of method `m()` is legal on line 5. The parent definition on line 8 has a transition to `self`, which is state `X` within the context of the definition on line 11. State `Y` is a substate of state `X`, therefore the definition is legal.

Enforcing the substitutability property for substates is important if the Hanoi model is used as part of a static analysis, where we may not know the exact state that an object is in at a particular point in time. We may, however, know “the current state is a substate of `X`” or “the current state is `X` or `Y`” and therefore at least be able to verify the safety of calls to methods allowed in state

X, or the methods in the least upper bound of X and Y. Without the subtyping relationship between child and parent states, we could not determine the safety of any method call without knowing the exact state the object is currently in.

3.1 Model Subtyping

In addition to a subtyping relationship between parent and child states within a particular model, Hanoi defines a subtyping relationship between models that mirrors the inheritance hierarchy of the types which are being modelled. If we have two types, T_1 and T_2 , where $T_2 <: T_1$, then the Hanoi model of T_2 must be a *simulation* of the Hanoi model of T_1 .

This relationship allows a child model to be more permissive than its parent, in that it can accept sequences of calls that are illegal in the parent. For instance, the `ByteArrayInputStream` class in Java supports marking, and if we know the object is a `ByteArrayInputStream` rather than an `InputStream`, we can safely allow calls to `mark()` without first checking that `markSupported()` returns true. One could also conceive of an iterator over an “infinite” sequence (such as a sequence of random numbers), where `hasNext()` would always return true and therefore unrestricted calls to `next()` are safe.

4 Runtime Implementation using Dynamic Proxies

Hanoi can be used at runtime, as a means of verifying that calls made to an interface with a Hanoi model are legal as it is being used. This is achieved in Java at present using a dynamic proxy — a runtime generated implementation of an interface which wraps the real implementation. 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.

This complementary approach to a static analysis can be useful in its own right. Flexibility in how the rules defined are checked and enforced can also be configurable. For instance, checking could include recording the last 10 method calls made before a failure, with the call stack for each call, to aid debugging in the event of failure. Enforcement could be loose, choosing to log a violation and cease monitoring rather than forcefully rejecting the method call. Providing such options to the developer allows for the usage of Hanoi in various contexts, such as passive debugging of live systems, to early detection of issues when unit testing.

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 must be done manually. We believe both issues can be overcome by using AspectJ [7]. This would allow us to inject the required checks directly into the classes, either at compile time or at runtime through a specialised class loader.

5 Future Work

Hanoi, in its current state, allows for the definition of models which support the majority of stateful interfaces we have encountered in the Java language. We believe this is a good foundation on which to build, though one key type of stateful behaviour is still unsupported: Exception transitions. The documentation of some types indicate that, upon throwing an exception, the object is no longer usable or must be re-initialised in some sense. This is common for `IOExceptions` on sockets or files, which signal the unexpected closure of the stream. Supporting this should be a straightforward extension of syntax to allow for exception conditions on transition definitions.

5.1 Static Analysis

Once we are confident Hanoi can express all the stateful behaviour we are interested in at this stage, we intend to investigate further the extent to which a static analysis could be performed on code which uses Hanoi modelled interfaces. Many of the techniques described in the literature to date rely on linear types. Static analysis becomes significantly more difficult once aliasing is permitted. We would like to investigate the requirements for aliasing on stateful objects in more detail, to better understand the types of sharing that occur.

One complimentary aspect of defining a stateful contract, as Hanoi does, is ensuring that the implementor also obeys the contract. In this case, this would mean ensuring that when the state machine indicates a method must be callable in a certain state, that we can prove this is always the case with a certain implementation. This is an active area of research, often involving automated theorem provers or other forms of relatively expensive static analysis. At present we have no plans to investigate this area further.

5.2 Parallel State Machines

We have observed in a number of stateful interfaces that there are often stateful groups of methods that can be modelled in isolation from the others. A good example of this can be seen in the Java `ListIterator` (which allows both forwards and backwards iteration) or on a bounded queue. Listing 1.5 shows a Hanoi model for a simple bounded queue, which allows items to be `dequeue()`'d, if elements are available, and `enqueue()`'d, if the queue is not full. This model is complex as these operations can be arbitrarily interleaved, which results in duplication as a consequence of the enforcement of transition narrowing. Alternatively, these two aspects of the queue could be modelled independently, as shown in Listing 1.6. Here, the “sections” declare submachines that operate in parallel and do not interact with one another. This eliminates the duplication and makes the model easier to understand. As the submachines cannot interact with one another however, some ‘information’ can be lost. The comments in Listing 1.5 indicate some subtle logic that can be exploited in the state transitions. If we have just successfully dequeued an element in state `NOT_EMPTY`, then there

Listing 1.5. A bounded queue model

```

1 NOT_EMPTY {
2   NOT_EMPTY_NOT_FULL {
3     enqueue(E) -> NOT_EMPTY
4     dequeue() -> NOT_FULL
5   }
6   dequeue() -> DEFAULT // could be NOT_FULL
7   isFull() :: false -> NOT_FULL_NOT_EMPTY
8 }
9
10 NOT_FULL {
11   NOT_FULL_NOT_EMPTY {
12     dequeue() -> NOT_FULL
13     enqueue(E) -> NOT_EMPTY
14   }
15   enqueue(E) -> DEFAULT // could be NOT_EMPTY
16   isEmpty() :: false -> NOT_EMPTY_NOT_FULL
17 }
18 isEmpty() :: false -> NOT_EMPTY
19 isEmpty()
20 isFull() :: false -> NOT_FULL
21 isFull()

```

must be space for at least one new element, meaning we could safely transition to `NOT_FULL`. Similarly, if we have just successfully enqueued an element in state `NOT_FULL`, there is at least one element in the list and therefore we could safely transition to `NOT_EMPTY`. Such interactions cannot be expressed in the parallel submachine case.

The submachines defined in Listing 1.6 also hint at one interesting possibility: as the submachines do not interact, they can be safely used from two separate aliases. One alias, restricted to the `CONSUMER` section could be given to one client, while another alias restricted to the `PRODUCER` section could be given to another. Method calls within the allocated section are safe. We intend to investigate this idea further, as part of understanding the implications of aliasing.

6 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.

Listing 1.6. A bounded queue modelled as two parallel machines

```

1 SECTION CONSUMER {
2   NOT_EMPTY { poll() -> DEFAULT }
3   isEmpty() :: false -> NOT_EMPTY
4   isEmpty()
5 }
6 SECTION PRODUCER {
7   NOT_FULL { enqueue(E) -> DEFAULT }
8   isFull() :: false -> NOT_FULL
9   isFull()
10 }

```

Acknowledgements The first author is supported by a studentship from the Scottish Informatics and Computing Science Alliance (SICSA) and the University of Glasgow. The work has also been supported by the EPSRC project “Engineering Foundations for 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 (University of Glasgow) for fruitful discussions related to subtyping in Hanoi.

References

1. J. Aldrich, J. Sunshine, D. Saini, and Z. Sparks. Typestate-Oriented Programming. In *OOPSLA*, pages 1015–1022. ACM, 2009.
2. K. Bierhoff, N.E. Beckman, and J. Aldrich. Practical API Protocol Checking with Access Permissions. In *ECOOP*, volume 5653 of *Lecture Notes in Computer Science*, pages 195–219, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.
3. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.
4. Simon J Gay, Vasco T Vasconcelos, Nils Gesbert, Alexandre Z Caldeira, and António Ravara. Modular Session Types for Distributed Object-Oriented Programming. *POPL*, 2010.
5. D. Harel. Statecharts: A visual formalism for complex systems. *Science of computer programming*, 8(3):231–274, 1987.
6. G. Hunt, J.R. Larus, M. Abadi, M. Aiken, P. Barham, M. Fahndrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, et al. An overview of the Singularity project. *Microsoft Research MSR-TR-2005*, 2005.
7. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W.G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, pages 327–353, 2001.
8. Robert E Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 12(1):157–171, 1986.