View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Institutional Research Information System University of Turin

Concurrent Typestate-Oriented Programming in Java

Rosita Gerbo Luca Padovani
Dipartimento di Informatica, Università di Torino, Italy

We describe a generative approach that enables concurrent typestate-oriented programming in Java and other mainstream languages. The approach allows programmers to implement objects exposing a state-sensitive interface using a high-level synchronization abstraction that synchronizes methods with the states of the receiver object in which those methods have an effect. An external tool takes care of generating all the boilerplate code that implements the synchronization logic. Behavioral types are used to specify object protocols. The tool integrates protocol conformance verification with the synchronization logic so that protocol violations are promptly detected at runtime.

1 Introduction

Beckman *et al.* [2] report that objects with a state-dependent interface are common in Java applications. Typical examples of such objects are *iterators* (which can be advanced only if they are not finished), *files* (which can be read or written only when they are open), and *locks* (which can be released only if they have been previously acquired). Implementing and using objects with a state-dependent interface is difficult and error prone, to the point that researchers have investigated specific methodologies – such as *typestate-oriented programming* (TSOP for short) [1, 10] – to help software development. TSOP is based on a set of programming abstractions supporting the design of objects with state-dependent interfaces and a behavioral type system ensuring that methods invoked on an object belong to the interface corresponding to that object's state. Crafa and Padovani [6] have extended TSOP to a concurrent setting, showing that the Objective Join Calculus [8] is a natural formal model for (concurrent) TSOP because of its built-in support for *join patterns*. In fact, join patterns make it possible not only to *explicitly associate methods with states* (which is one of the disinguishing features of TSOP as conceived by Aldrich *et al.* [1]) but also to *synchronize methods and states*: the process invoking a method that is not available in an object's current state is suspended until the object moves into a state for which that method is available again.

While the feasibility of Crafa and Padovani's approach to concurrent TSOP is partially witnessed by a proof-of-concept type checker for the Objective Join Calculus [18, 17], applying it to mainstream programming languages presents two substantial problems. (P1) The few languages that feature built-in join patterns – notably JoCaml [9], Cω [3, 16], Join Java [14] and JErlang [20] – are mostly experimental languages serving specialized communities and/or have not been maintained for a long time. Library implementations of join patterns [21, 22, 13, 24, 27] have similar issues. (P2) Retrofitting Crafa and Padovani's type system into an existing language is conceptually and technically challenging. The ongoing efforts on the implementation of Linear Haskell [4] show that this is the case even when retrofitting a streamlined substructural type system in the controlled setting of a pure functional language.

The main contribution of this work is a practical approach that makes concurrent TSOP immediately applicable to Java and, in fact, to virtually every programming language. The approach is necessarily based on compromises. We address problem (P1) using *code generation*: the programmer writing a Java class using our approach specifies the join patterns that synchronize states and methods by means of *standard Java annotations* [11]; an external tool generates the boilerplate code that implements all the synchronization logic. As a result, we avoid any dependency on join patterns in the programming

F. Martins, D. Orchard (Eds.): Programming Language Approaches
to Concurrency- & Communication-cEntric Software (PLACES 2019)
EPTCS 291, 2019, pp. 24–34, doi:10.4204/EPTCS.291.3

© Rosita Gerbo and Luca Padovani
This work is licensed under the
Creative Commons Attribution License.

language or in external libraries. We address problem (P2) trading *compile-time* with *runtime* protocol conformance verification. The programmer provides a protocol specification for the class, again in the form of a Java annotation, and the same external tool generates the code that detects all protocol violations at runtime if (and as soon as) they actually occur. This way, we bypass the non-trivial problem of reconciling fundamentally different type systems, at the cost of delayed detection of programming errors. From a technical standpoint, we also contribute a refinement of Le Fessant and Maranget's [7] compilation scheme for join patterns that uses behavioral types to identify object protocol violations and to prune the state space of the automaton – called *matching automaton* – that performs join pattern matching.

The rest of the paper is organized as follows. We start recalling the key features of concurrent TSOP with join patterns [6] through a simple example of concurrent object (Section 2). Then, we describe the construction of the matching automaton (Section 3), which is instrumental to the subsequent code generation phase (Section 4). We conclude in Section 5.

The tool has been implemented and used to generate all the code and automata shown in the paper. Its source code is publicly available [19].

2 Example

We illustrate the basics of concurrent TSOP in the Objective Join Calculus [8, 6] by modeling a *promise* or *completable future variable* (the terminology varies depending on the language). We can think of a completable future variable as a one-place buffer that must be written (or completed) once (this is the view of the variable that is sometimes called *promise* [12]) and that can be read any number of times, concurrently. We find this specific example appropriate because it is a relatively simple concurrent object that exposes a state-sensitive interface and whose protocol uses all the connectives of the type language we are about to discuss. Below is the definition of a *future* object of type E_{future}:

$$\begin{array}{l} \texttt{object}\ \mathit{future} : \mathsf{E}_{\mathit{future}}\ [\ \texttt{EMPTY}\ \&\ \texttt{put}(x) \triangleright \mathit{future}!\texttt{FULL}(x) \\ \qquad\qquad\qquad\quad \mid \texttt{FULL}(x)\ \&\ \texttt{get}(\mathit{user}) \triangleright \mathit{future}!\texttt{FULL}(x)\ \&\ \mathit{user}!\texttt{reply}(x)\ ] \\ \mathit{future}!\texttt{EMPTY} \end{array} \tag{1}$$

Messages sent to the *future* object are stored into its *mailbox*. The object understands four kinds of messages, each identified by a *tag*: `EMPTY` and `FULL` model the state of the object while `put` and `get` its operations. Thus, an `EMPTY` message in the *future*'s mailbox (denoted by a term $\mathit{future}!\texttt{EMPTY}$) means that *future* has not been completed yet, whereas a $\texttt{FULL}(x)$ message in the *future*'s mailbox means that *future* has been completed with value x. The behavior of the object is given by the two reactions $J \triangleright P$ within brackets. When (some of) the messages in the object's mailbox match the join pattern J of a reaction, the matched messages are atomically consumed and those on the right hand side of the reaction are produced. Above, the first reaction specifies that a *future* in state `EMPTY` accepts a `put` operation carrying an x argument and changes its state to $\texttt{FULL}(x)$. The second reaction specifies that a *future* in state $\texttt{FULL}(x)$ accepts a `get` operation carrying an argument *user*. The reaction leaves the state of the object unchanged (the message $\texttt{FULL}(x)$ is restored into the *future*'s mailbox) and additionally stores $\texttt{reply}(x)$ into *user*'s mailbox, from which the user of the completable future variable can retrieve the value of x. The last line of the definition (1) acts as a constructor that initializes *future* to state `EMPTY`.

As defined, *future* does not react to message patterns of the form $\texttt{EMPTY}\ \&\ \texttt{get}(\mathit{user})$ or $\texttt{FULL}(x)\ \&\ \texttt{put}(y)$. After all, an uncompleted future variable cannot provide its value and a completed future variable should not be completed again. Nonetheless, the two scenarios differ crucially: trying to read

an uncompleted future variable is *alright* (simply, the request will remain pending until the variable is completed), whereas trying to complete a future variable twice is just *wrong*. To tell the two cases apart we need a protocol specification for completable future variables. The protocols (or types) E, F we consider are generated by the grammar

$$\mathsf{E},\mathsf{F} \ ::= \ \mathbb{0} \ | \ \mathbb{1} \ | \ \mathtt{m} \ | \ *\mathtt{m} \ | \ \mathsf{E}+\mathsf{F} \ | \ \mathsf{E}\cdot\mathsf{F}$$

and are akin to regular expressions defined over tags $\mathtt{m}$, except that iteration is limited to single tags and '$\cdot$' is shuffling. Formally, we define the semantics of a protocol as the set of strings of tags inductively generated by the equations below

$$\begin{array}{lll} [\![\mathbb{0}]\!] = \emptyset & [\![\mathtt{m}]\!] = \{\mathtt{m}\} & [\![\mathsf{E}+\mathsf{F}]\!] = [\![\mathsf{E}]\!] \cup [\![\mathsf{F}]\!] \\ [\![\mathbb{1}]\!] = \{\varepsilon\} & [\![*\mathtt{m}]\!] = \{\mathtt{m}^n \mid n \in \mathbb{N}\} & [\![\mathsf{E}\cdot\mathsf{F}]\!] = \{u_1 v_1 \cdots u_n v_n \mid u_1 \cdots u_n \in [\![\mathsf{E}]\!], v_1 \cdots v_n \in [\![\mathsf{F}]\!]\} \end{array}$$

where ε denotes, as usual, the empty string, $\mathtt{m}^n$ is the string made of n occurrences of $\mathtt{m}$, and we write u_i and v_j to denote arbitrary, possibly empty strings of tags. A type specifies the legal ways of using an object. In particular, the only legal way of using an object of type $\mathtt{m}$ is to send an $\mathtt{m}$-tagged message to it, whereas an object of type $*\mathtt{m}$ can be sent an arbitray number of $\mathtt{m}$ messages. An object of type $\mathsf{E}+\mathsf{F}$ can be used either as specified by E or as specified by F, whereas an object of type $\mathsf{E}\cdot\mathsf{F}$ must be used as specified by both E and F. For example, an object of type $\mathtt{EMPTY}\cdot\mathtt{put}$ must be sent both an `EMPTY` message and also a `put` message, in whichever order. The constants $\mathbb{0}$ and $\mathbb{1}$ have a subtle semantics. The only legal way of using an object with type $\mathbb{1}$ is *not using it*, because $[\![\mathbb{1}]\!]$ contains the empty string only. Concerning $\mathbb{0}$, there is no legal way of using an object with that type, because $[\![\mathbb{0}]\!]$ is empty. Even *not using* such an object is illegal! This seemingly bizarre interpretation of $\mathbb{0}$ is actually key in the construction of the matching automaton, as we will see in Section 3. In the following, we write $\mathsf{E}=\mathsf{F}$ if $[\![\mathsf{E}]\!] = [\![\mathsf{F}]\!]$. In particular, $\mathsf{E}\cdot\mathsf{F} = \mathsf{F}\cdot\mathsf{E}$. By using *shuffling* instead of sequential composition we deprive types of any information concerning the *order* in which messages are supposed to be sent to objects. There are two motivations for this choice. First, it is often impossible to determine the order of messages sent to an object by concurrent (hence, independent) processes. Shuffling allows us to easily overapproximate an object protocol while ignoring any ordering constraint. Second, we will use types in the construction of the matching automaton, whose main purpose is to detect when a reaction can fire. According to the semantics of the Objective Join Calculus [8], only the *presence* – not the order – of messages in the mailbox matters in this respect.

As an example of object type, we define $\mathsf{E}_{\mathit{future}}$ thus:

$$\mathsf{E}_{\mathit{future}} \stackrel{\text{def}}{=} *\mathtt{get}\cdot(\mathtt{EMPTY}\cdot\mathtt{put}+\mathtt{FULL}) \tag{2}$$

This type specifies that a completable future variable is always either `EMPTY` or `FULL` (*cf.* the '$+$' connective), that it can be completed once when it is `EMPTY` (*cf.* the innermost '$\cdot$' connective), and that it may receive an arbitrary number of `get` messages (*cf.* the '$*$' connective) regardless of its state (*cf.* the outermost '$\cdot$' connective, allowing any interleaving of the `get` messages with respect to all the other messages).

We assume a well-formedness condition on types, requiring that every starred tag occurring in them does not occur also unstarred. This assumption simplifies the technical development and does not appear to impact the expressiveness of types in describing interesting object protocols.

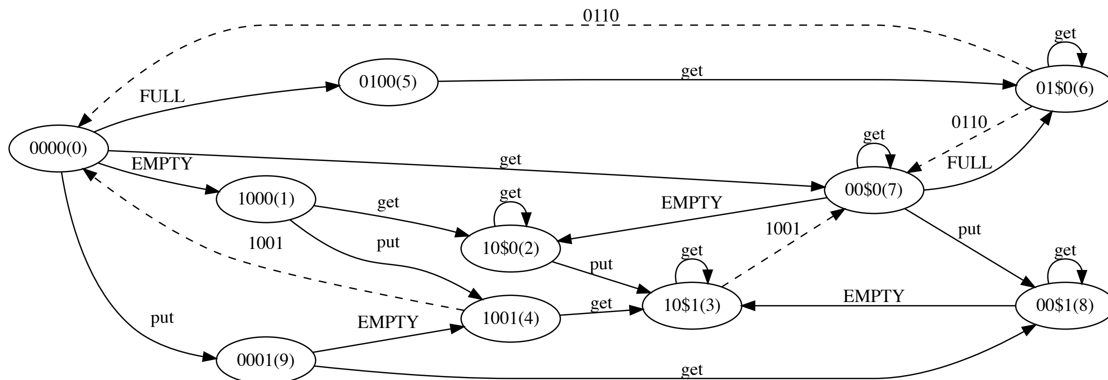

Figure 1: Matching automaton for the join patterns of the future variable.

3 From join patterns to matching automata

In this section we describe a compilation scheme from join patterns to *matching automata*. A matching automaton is a particular kind of finite-state automaton (FSA) whose purpose is to efficiently detect when one of the object's reactions can fire and if a protocol violation has occurred. To do so, the matching automaton keeps track of the state of the object's mailbox as messages are stored in it. The compilation scheme we are about to illustrate is close to the one given by Le Fessant and Maranget [7], except that we use the behavioral type associated with a join pattern to prune the state space of the resulting FSA. We illustrate the compilation scheme using the definition of *future* (1) as guiding example. The resulting matching automaton is shown in Figure 1.

From the join pattern being compiled we determine a *signature* Σ of message tags, those understood by the object, which is also the alphabet of the resulting matching automaton. The signature usually coincides with the set of tags occurring in the type of the object. In the case of *future* we have $\Sigma = \{\mathtt{EMPTY}, \mathtt{FULL}, \mathtt{get}, \mathtt{put}\}$. Tags are assumed to be *totally ordered*, for instance lexicographically.

The *states* of the automaton are n-tuples of *counters* $\alpha_1 \cdots \alpha_n$ that provide an approximate description of the content of the object's mailbox. The length n of the tuple is the cardinality of Σ and the i-th counter α_i of the tuple is an element of $\mathbb{N} \cup \{\$\}$ that corresponds to the i-th tag $\mathtt{m}_i \in \Sigma$ according to the tag total order. A counter $\alpha_i \in \mathbb{N}$ means that there are *exactly* α_i messages with tag $\mathtt{m}_i$ in the mailbox, whereas a counter $\alpha_i = \$$ means that there is *at least* one message with tag $\mathtt{m}_i$ in the mailbox. We use the behavioral type of the object to determine the form of counters. We say that a message $\mathtt{m}$ is *unbounded* in E if $*\mathtt{m}$ occurs in E and that it is *bounded* otherwise. It is easy to show that, for every bounded message $\mathtt{m}$ in E, there exists $N \in \mathbb{N}$ such that every trace in $[\![\mathsf{E}]\!]$ contains at most N occurrences of $\mathtt{m}$, in which case we say that $\mathtt{m}$ is N-bounded. We choose counters of the form $k \in \{0, \dots, N\}$ for N-bounded messages and of the form 0 or \$ for unbounded messages. As we will see shortly, the distinction between bounded and unbounded messages is needed for handling the automaton's transitions. In E_{future}, $\mathtt{EMPTY}$, $\mathtt{FULL}$ and $\mathtt{put}$ are 1-bounded and $\mathtt{get}$ is unbounded. The *initial state* of the automaton is $0 \cdots 0$, describing an empty mailbox. For ease of reference, the states in Figure 1 are followed by a unique index in parentheses.

The automaton has two kinds of transitions. *Receive transitions* (the solid arrows in Figure 1) correspond to the arrival of a message in the object's mailbox. For example, the future's automaton has an

EMPTY-tagged transition from state 0000 to state 1000 recording the fact that, if the future's mailbox is empty and an EMPTY message arrives, we end up in a mailbox configuration that has exactly one EMPTY message and no other message. *Consume transitions* (the dashed arrows in Figure 1) correspond to the firing of a reaction, which causes the consumption of the messages occurring in the join pattern from the object's mailbox. Consume transitions are slightly more difficult to handle than receive transitions. The residual number of bounded messages after a consume transition can be computed exactly by decrementing the corresponding counters in the starting state. On the contrary, the counter describing the residual number of an unbounded message after a consume transition can be 0 or $ depending on whether all such messages have been consumed or if at least one remains in the mailbox. For example, in Figure 1 there are two such transitions from state 01$0 leading to states 0000 and 00$0 depending on whether the consumed get message was the last one with such tag in the mailbox or not. The implementation of the automaton deterministically chooses which consume transition to follow by means of a runtime check. The fewer unbounded messages there are in a type, the fewer runtime checks are necessary, the more efficient the code of the matching automaton is (more on this in Section 4). We label consume transitions with an n-tuple indicating which messages are consumed, hence the reaction that fires.

This basic construction admits a few refinements which help reducing the size of the resulting FSA. For example, Le Fessant and Maranget [7] observe that, under the assumption that a reaction fires *as soon as* possible, receive transitions departing from states with outgoing consume transitions are useless. Indeed, as the messages consumed by the reaction are removed from the mailbox, the automaton will move to a state describing a mailbox configuration with *strictly fewer* messages. As we will see in Section 4, this refinement is not applicable in our setting because our implementation of the matching automaton does not always guarantee such prompt firing of transitions. Another refinement not considered by Le Fessant and Maranget [7] is made possible by the use of behavioral types. The attentive reader may have noticed that there is no state corresponding to the tuple 1100 in the FSA of Figure 1. This is not an oversight: the tuple 1100 describes a mailbox configuration containing *both* an EMPTY message *and also* a FULL message, but this configuration is not found in $[\![\mathsf{E}_{future}]\!]$. If such configuration is reached, the object should notify the user (*e.g.* through an exception) of the fact that its protocol has been violated rather than trying to handle the situation in some way.

To identify *illegal states* we need an operator on types that is closely related to *Brzozowski derivative* for plain regular expressions [5] and that is inductively defined thus:

$$\begin{array}{l}\mathbb{0}[\mathtt{m}] = \mathbb{0} \\ \mathbb{1}[\mathtt{m}] = \mathbb{0}\end{array} \quad \mathtt{m}[\mathtt{m}'] = \begin{cases}\mathbb{1} & \text{if } \mathtt{m} = \mathtt{m}' \\ \mathbb{0} & \text{otherwise}\end{cases} \quad (*\mathtt{m})[\mathtt{m}'] = \begin{cases}*\mathtt{m} & \text{if } \mathtt{m} = \mathtt{m}' \\ \mathbb{0} & \text{otherwise}\end{cases} \quad \begin{array}{r}(\mathsf{E}+\mathsf{F})[\mathtt{m}] = \mathsf{E}[\mathtt{m}] + \mathsf{F}[\mathtt{m}] \\ (\mathsf{E}\cdot\mathsf{F})[\mathtt{m}] = \mathsf{E}[\mathtt{m}]\cdot\mathsf{F} + \mathsf{E}\cdot\mathsf{F}[\mathtt{m}]\end{array}$$

In words, $\mathsf{E}[\mathtt{m}]$ describes the language of traces obtained by removing a single occurrence of m from those traces of E that have at least one. Note that $\mathsf{E}[\mathtt{m}] = \mathbb{0}$ if no trace of E contains at least an occurrence of m. As we will see shortly, this property is key for identifying illegal states.

The next step is to annotate each state of the FSA with a type, as follows: the initial state is annotated with the type assigned to the object by the programmer. If a state s_1 annotated with E has an outgoing transition labeled m to another state s_2, then s_2 is annotated with $\mathsf{E}[\mathtt{m}]$. Since there may be distinct paths leading to the same state in the FSA, one may wonder whether this annotation procedure is well defined. It is possible to show that the annotation for each state is unique modulo type equivalence. This follows from three facts: (1) the FSA is constructed in such a way that distinct minimal paths (of receive transitions) between two states only differ for the order of tags, (2) the order of derivatives is irrelevant, namely $\mathsf{E}[\mathtt{m}][\mathtt{m}'] = \mathsf{E}[\mathtt{m}'][\mathtt{m}]$ for all E, m and m′ and (3) type well-formedness ensures that $\mathsf{E}[\mathtt{m}] = \mathsf{E}$ if m is unbounded in E. To illustrate, let us annotate a few states of Figure 1. The initial state is annotated with E_{future}. From this state we can reach the state 1000 with an EMPTY-labeled transition, hence we annotate

```
1  @Protocol("*get·(EMPTY·put + FULL)")
2  class Future<A> {
3    @State     private void EMPTY();
4    @State     private void FULL(A x);
5    @Operation public  A    get();
6    @Operation public  void put(A x);
7    @Reaction  private void when_EMPTY_put(A x) { this.FULL(x); }
8    @Reaction  private A    when_FULL_get(A x) { this.FULL(x); return x; }
9    public Future() { this.EMPTY(); }
10 }
```

Figure 2: Java implementation of a completable future variable (source code).

1000 with $\mathsf{E}_{future}[\texttt{EMPTY}] = *\texttt{get} \cdot \texttt{put}$. From 1000 we can reach the state 1001 with a put-labeled transition, hence we annotate 1001 with $\mathsf{E}_{future}[\texttt{EMPTY}][\texttt{put}] = (*\texttt{get} \cdot \texttt{put})[\texttt{put}] = *\texttt{get}$. Note that there is another path from 0000 to 1001 labeled with put, EMPTY and that $\mathsf{E}_{future}[\texttt{put}][\texttt{EMPTY}] = *\texttt{get}$. Now consider the state 1100. This state could be reached from 0000 along a path labeled with EMPTY, FULL. We compute $\mathsf{E}_{future}[\texttt{EMPTY}][\texttt{FULL}] = (*\texttt{get} \cdot \texttt{put})[\texttt{FULL}] = \mathbb{0}$. The fact that we have obtained $\mathbb{0}$ means that the state 1100 describes a configuration of the mailbox that violates the type E_{future}. In general, every state for which the type annotation is (equivalent to) $\mathbb{0}$ is *illegal* and can be eliminated. Figure 1 shows the FSA without illegal states.

The elimination of illegal states is not just an optimization aimed at reducing the size of the resulting FSA, but plays a key role for the soundness of our approach, which is based on a mixture of compile-time and runtime checks. We have stated that the matching automaton should recognize all and only those mailbox configurations that are legal according to the behavioral type of the object so that, in case or protocol violation, a suitable exception can be thrown. With the above construction, a protocol violation is promptly detected as a missing receive transition from the current state of the FSA.

4 Code generation

A Java programmer using our concurrent TSOP approach for implementing a completable future variable writes the code shown in Figure 2. What we see there is a syntactically-valid Java class with a few Java annotations (in magenta) that are specific to our approach. All the boilerplate code that stores incoming messages, matches join patterns and watches for protocol violations is automatically generated from this code. Before looking at the generated code, we trace the correspondence between the Java code in Figure 2 and the Objective Join Calculus constructs in (1).

The `@Protocol` annotation on line 1 specifies the behavioral type associated with the class and corresponds to the type E_{future} in (1). As we have discussed in Section 3, this type is necessary to build the matching automaton corresponding to the join patterns in the class. The type refers to the name of the Java methods declared on lines 3–6, which correspond to messages in the Objective Join Calculus. Sending a message to an object in the Objective Join Calculus amounts to invoking the corresponding method in Java where the arguments of the method are those of the message, with some exceptions discussed shortly. The body of these methods will be generated automatically.

Methods corresponding to messages have either a `@State` or an `@Operation` annotation. Methods of the first kind, such as EMPTY and FULL, model state messages from which we do not expect to receive

an answer. For this reason, the return type of state methods is always `void`. Methods of the second kind, such as `get` and `put`, model operations on the object from which we (usually) expect to receive an answer (possibly just a signal meaning that the operation is complete). In the Objective Join Calculus, messages like `get` carry an explicit continuation that the object uses to answer the request. The Java idiom to communicate results is through returned values instead of explicit continuations. For this reason, operation methods usually have one less argument compared to the corresponding message and their return type is different from `void`, as in the case of `get`. The `put` method is a notable exception. The corresponding message does not have a continuation argument because no answer is expected from a `put` message. However, our code generator relies on the fact that each join pattern combines *exactly one* operation method and zero or more state methods. Because of this requisite (also found in Cω [3]), `put` is qualified as an operation method even if does not return any significant result. The `void` return type is the obvious choice for operation methods like `put` that do not return anything.

Methods defined on lines 7–8 have a `@Reaction` annotation, meaning that they correspond to reactions in the Objective Join Calculus. The name of a reaction method reveals the structure of the join pattern it represents and is built from the `when` prefix followed by the tags of the messages in the join pattern separated by underscores. So, the method names `when_EMPTY_put` and `when_FULL_get` respectively correspond to the join patterns `EMPTY & put(`*x*`)` and `FULL(`*x*`) & get(`*user*`)`. The argument list of a reaction method is the concatenation of the argument lists of the joined messages (without explicit continuations), whereas its return type is that of the one and only operation method that appears in the pattern. For example, `when_FULL_get` has a single argument (coming from `FULL`) and return type `A` (the same of `get`). The body of a reaction method is the Java transposition of the corresponding process in the Objective Join Calculus reaction. The body of `when_EMPTY_put` changes the state of the object to `FULL` by invoking the corresponding method. The body of `when_FULL_get` restores the state of the object to `FULL` and then `return`s the content of the future variable to the client. The constructor (line 9) is unremarkable.

We use Java access modifiers to control the visibility of methods: methods corresponding to state messages are `private` to enforce the fact that, as in the original presentation of TSOP [1], state transitions can only be triggered from within the class; methods corresponding to operations are `public`, for these provide the public interface to the object; reaction methods are `private`, since they will be invoked by the automatically generated code that fires reactions (more on them later).

We now illustrate how our generator expands the source class in Figure 2 into a fully functional class. Because of space limitations, we can only focus on a few bits of generated code, shown in Figure 3.

First of all, the generator adds some fields (lines 1–6) to enforce exclusive access to instances of the class, to represent the state of the matching automaton, and to represent message queues. Mutual exclusion is guaranteed by a reentrant `lock` (line 1) and by condition variables associated with the operation methods (`try_get` and `try_put` on lines 2–3). It would also be possible to use implicit locks and the built-in synchronization facilities of Java, but using explicit locks is more flexible and sometimes results in better performing code. The `state` of the FSA is represented as a field of type `int` that contains the indexes shown within parentheses in Figure 1. Its initial value 0 corresponds to the initial state of the FSA. The concrete representation of message queues depends on whether messages are bounded and/or have arguments. Bounded messages without arguments do not need a queue, for all we need to know is their number in the mailbox and this number is encoded accurately in the state of the FSA. For this reason, there are no fields corresponding to `EMPTY` and `put` messages. For unbounded message without arguments the queue is just a counter field of type `int` that tracks the number of those messages in the mailbox. Note that `get` is one of such messages (line 6). Indeed, the argument of the `get` message is an explicit continuation that disappears in Java. For 1-bounded messages with a single argument the queue

```
 1 private ReentrantLock lock;
 2 private Condition try_get;
 3 private Condition try_put;
 4 private int state = 0;
 5 private A queue_FULL = null;
 6 private int queue_get = 0;
 7
 8 private void FULL (A x) {
 9   lock.lock();
10   queue_FULL = x;
11   switch (state) {
12   case 0:
13     state = 5;
14     lock.unlock();
15     break;
16   case 7:
17     state = 6;
18     try_get.signal();
19     lock.unlock();
20     break;
21   default:
22     lock.unlock();
23     throw new IllegalStateException();
24   }
25 }
```

```
26 public A get () {
27   lock.lock();
28   queue_get++;
29   switch (state) {
30   case 0: state = 7; break;
31   case 1: state = 2; break;
32   case 4: state = 3; break;
33   case 5: state = 6; break;
34   case 9: state = 8; break;
35   default: break;
36   }
37   while (true)
38     switch (state) {
39     case 6: {
40       final A x = queue_FULL;
41       queue_FULL = null;
42       queue_get--;
43       state = queue_get == 0 ? 0 : 7;
44       lock.unlock();
45       return when_FULL_get(x);
46     }
47     default:
48       try_get.awaitUninterruptibly();
49     }
50 }
```

Figure 3: Java implementation of a future variable (generated code).

coincides with the value of the argument (whether the message is present or not is encoded in the state of the FSA). This is the case of `FULL`, whose field has the same type as its argument (line 5). In all the other cases we use a real queue. There is no message with such properties in the example we are considering, hence no example of such field. Analogous optimized representations for message queues have been described in the literature [8, 27].

We now turn the attention to the generated body of method `FULL`, bearing in mind that invoking this method means sending a message `FULL` to the object. The method enters the critical section (line 9) and stores its argument in the corresponding queue (line 10). The following `switch` implements the transitions of the FSA by analysing and updating the `state` field. There are two receive transitions labeled `FULL` in Figure 1. The first one (lines 12–15) moves the FSA from state 0 to state 5. Since 5 is not a firing state (it has no outgoing consume transitions), we just leave the critical section and quit the method. The second transition (lines 16–20) moves the FSA to the firing state 6. If this happens, another process has previously invoked `get` and is waiting for the future variable to be resolved. Thus, we notify the condition variable `try_get` to wake such process before leaving the critical section. Note that, after the notification and before the awoken process is scheduled to run, other processes may invoke a method (*e.g.* `get`) on the object and successfully enter the criticial section. For this reason, our compilation technique differs from the one of Le Fessant and Maranget [7] in that it does not guarantee that reactions fire *as soon as possible*. To conclude the description of the `FULL` method, the `default` case of the `switch` deals with protocol violations (lines 21–23). In this case, the `FULL` method has been invoked in a state in which this message was not expected to arrive. The violation is notified with an exception

thrown just outside the critical section (line 23).

The code generated for operation methods is more complex, because it involves not only updating the state of the FSA, but also detecting and executing reactions that fire. Let us have a look at the `get` method, whose body is divided into two parts. The first part (lines 27–36) stores the `get` message (by incrementing the corresponding counter) and updates the state of the FSA. This part is similar in structure to the generated body of state methods like `FULL`, except for two differences. First, the `default` case (line 35) corresponds to the loops in Figure 1, which do not change the state of the FSA. In particular, the behavioral type that we are considering allows `get` messages to arrive at any time, hence `get` messages are never a cause of protocol violation, unlike `FULL` messages. Second, no condition variable is notified, even if the FSA ends up in a firing state, because it is the very process executing this code that takes care of executing the code corresponding to the reaction (*cf.* line 33).

The second half of the method contains the code that detects and executes a firing reaction (lines 37–49). The state of the FSA is repeatedly checked: as long as the state is a non-firing one, the method suspends and waits to be awoken (line 48); when a firing state is detected (line 39), the messages consumed by the firing reaction are removed from the corresponding queues and their arguments (if there are any) are stored in local variables (lines 40–42). The state of the FSA is then updated to account for the removal of messages from the mailbox of the object (line 43). Unlike previous state updates, this one requires a runtime check: in Figure 1, there are two outgoing consume transitions from the firing state 6, depending on whether the residual number of `get` messages in the mailbox is 0 or not. In the first case, the FSA goes back to the initial state. In the second case, the FSA moves to state 7. Finally, the method leaves the critical section (line 44) and invokes the appropriate reaction method (line 45).

5 Concluding Remarks

We have presented a generative approach enabling concurrent TSOP in Java, thus providing a practical implementation of the concurrent TSOP methodology whose theoretical foundations have been studied in previous work [6]. Our approach makes very few assumptions on the host programming language and is therefore easily portable to other mainstream languages. *En passant*, we have described a refinement of Le Fessant and Maranget's [7] compilation scheme for join patterns using behavioral types and have shown an implementation technique of join patterns that integrates smoothly with – and takes advantage of – Java without relying on libraries or language extensions. Specifically, we use the native sequential composition and `return` instead of explicit continuations for imposing an order in the execution of code and for returning the result of operations.

Compared to the theoretical study of Crafa and Padovani [6], here we have adopted a simpler type language where only the tag of messages – and not the type of their arguments – is reported in an object protocol. This simplification is made possible by the fact that our approach defers typestate checking at runtime, thus rendering the type of message arguments unimportant. Besides, the use of sequential composition in place of explicit continuations reduces the need of message arguments with complex (behavioral) types and results in a more natural programming style. Another difference concerns the semantics of types, which is given here in terms of traces instead of multisets [6]. Nonetheless, the two semantics induce the very same notion of type equivalence.

Plaid [25, 26] and StMungo [15] are notable implementations of TSOP. Plaid supports TSOP natively, whereas StMungo relies on external tools for analysing annotated Java code. The main advantage of Plaid and StMungo compared to our approach is that they are able to provide static protocol conformance guarantees, whereas our approach shifts most of the typestate checking at runtime. However, neither

Plaid nor StMungo support *concurrent* TSOP.

Many ideas for further developments stem from this work. For example, the runtime information on the state of the matching automaton could enable forms of error recovery that notoriously clash with static analysis. The same information might also be useful to implement a *gradual type system* for concurrent TSOP [23, 28]. Finally, we are aware that lock-based compilation of join patterns *à la* Le Fessant and Maranget [7] does not scale well to large numbers of processes. Unfortunately, scalable implementations of join patterns [27] are not based on matching automata, making it difficult to detect protocol violations. Whether scalability and runtime protocol checking can be reconciled remains to be established.

Acknowledgments. We are grateful for the valuable and detailed feedback received from the anonymous reviewers. This work has been partially supported by MSCA-RISE-2017 778233 BEHAPI.

References

[1] Jonathan Aldrich, Joshua Sunshine, Darpan Saini & Zachary Sparks (2009): *Typestate-oriented programming*. In: *Companion to OOPSLA'09*, ACM, pp. 1015–1022, doi:10.1145/1639950.1640073.

[2] Nels E. Beckman, Duri Kim & Jonathan Aldrich (2011): *An Empirical Study of Object Protocols in the Wild*. In: *Proceedings of ECOOP'11*, LNCS 6813, Springer, pp. 2–26, doi:10.1007/978-3-642-22655-7_2.

[3] Nick Benton, Luca Cardelli & Cédric Fournet (2004): *Modern concurrency abstractions for C#*. *ACM Transactions on Programming Languages and Systems* 26(5), pp. 769–804, doi:10.1145/1018203.1018205.

[4] Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones & Arnaud Spiwack (2018): *Linear Haskell: practical linearity in a higher-order polymorphic language*. *PACMPL* 2(POPL), pp. 5:1–5:29, doi:10.1145/3158093.

[5] Janusz A. Brzozowski (1964): *Derivatives of Regular Expressions*. *Journal of ACM* 11(4), pp. 481–494, doi:10.1145/321239.321249.

[6] Silvia Crafa & Luca Padovani (2017): *The Chemical Approach to Typestate-Oriented Programming*. *ACM Transactions on Programming Languages and Systems* 39, pp. 13:1–13:45, doi:10.1145/3064849.

[7] Fabrice Le Fessant & Luc Maranget (1998): *Compiling Join-Patterns*. *Electr. Notes Theor. Comput. Sci.* 16(3), pp. 205–224, doi:10.1016/S1571-0661(04)00143-4.

[8] Cédric Fournet, Cosimo Laneve, Luc Maranget & Didier Rémy (2003): *Inheritance in the Join Calculus*. *Journal of Logic and Algebraic Programming* 57(1-2), pp. 23–69, doi:10.1016/S1567-8326(03)00040-7.

[9] Cédric Fournet, Fabrice Le Fessant, Luc Maranget & Alan Schmitt (2003): *JoCaml: A Language for Concurrent Distributed and Mobile Programming*. In: *Lecture Notes of the 4th International School on Advanced Functional Programming (AFP'03)*, LNCS 2638, Springer, pp. 129–158, doi:10.1007/978-3-540-44833-4_5.

[10] Ronald Garcia, Éric Tanter, Roger Wolff & Jonathan Aldrich (2014): *Foundations of Typestate-Oriented Programming*. *ACM Transactions on Programming Languages and Systems* 36(4), p. 12, doi:10.1145/2629609.

[11] James Gosling, Bill Joy, Guy L. Steele, Gilad Bracha & Alex Buckley (2014): *The Java Language Specification, Java SE 8 Edition*, 1st edition. Addison-Wesley Professional.

[12] Philipp Haller, Aleksandar Prokopec, Heather Miller, Viktor Klang, Roland Kuhn & Vojin Jovanovic (2019): *Futures and Promises*. Available at `https://docs.scala-lang.org/overviews/core/futures.html`.

[13] Philipp Haller & Tom Van Cutsem (2008): *Implementing Joins Using Extensible Pattern Matching*. In: *Proceedings of COORDINATION'08*, LNCS 5052, Springer, pp. 135–152, doi:10.1007/978-3-540-68265-3_9.

[14] G. Stewart Von Itzstein & Mark Jasiunas (2003): *On Implementing High Level Concurrency in Java*. In: *Proceedings of ACSAC'03*, LNCS 2823, Springer, pp. 151–165, doi:10.1007/978-3-540-39864-6_13.

[15] Dimitrios Kouzapas, Ornela Dardha, Roly Perera & Simon J. Gay (2018): *Typechecking protocols with Mungo and StMungo: A session type toolchain for Java*. Science of Computer Programming 155, pp. 52 – 75, doi:10.1016/j.scico.2017.10.006.

[16] Microsoft Research (2004): *Cω*. Available at `https://www.microsoft.com/en-us/research/project/comega/`. Last accessed on January 2019.

[17] Luca Padovani (2017): *Cobalt***Blue** – *Behavioral Type Checking for Concurrent Objects*. Available at `http://www.di.unito.it/~padovani/Software/CobaltBlue/index.html`.

[18] Luca Padovani (2018): *Deadlock-Free Typestate-Oriented Programming*. Programming Journal 2, p. article 15, doi:10.22152/programming-journal.org/2018/2/15.

[19] Luca Padovani (2019): *EasyJoin – Concurrent TypeState-Oriented Programming in Java (Version 1.0)*, doi:10.5281/zenodo.2586829.

[20] Hubert Plociniczak & Susan Eisenbach (2010): *JErlang: Erlang with Joins*. In: Proceedings of COORDINATION'10, LNCS 6116, Springer, pp. 61–75, doi:10.1007/978-3-642-13414-2_5.

[21] Claudio V. Russo (2007): *The Joins Concurrency Library*. In: Proceedings of PADL'07, LNCS 4354, Springer, pp. 260–274, doi:10.1007/978-3-540-69611-7_17.

[22] Claudio V. Russo (2008): *Join Patterns for Visual Basic*. In: Proceedings of OOPSLA'08, ACM, pp. 53–72, doi:10.1145/1449764.1449770.

[23] Jeremy G. Siek & Walid Taha (2007): *Gradual Typing for Objects*. In: Proceedings of ECOOP'07, LNCS 4609, Springer, pp. 2–27, doi:10.1007/978-3-540-73589-2_2.

[24] Martin Sulzmann, Edmund S. L. Lam & Peter Van Weert (2008): *Actors with Multi-headed Message Receive Patterns*. In: Proceedings of COORDINATION'08, LNCS 5052, Springer, pp. 315–330, doi:10.1007/978-3-540-68265-3_20.

[25] Joshua Sunshine, Karl Naden, Sven Stork, Jonathan Aldrich & Éric Tanter (2011): *First-class state change in plaid*. In Cristina Videira Lopes & Kathleen Fisher, editors: Proceedings of OOPSLA'11, ACM, pp. 713–732, doi:10.1145/2048066.2048122.

[26] Joshua Sunshine, Sven Stork, Karl Naden & Jonathan Aldrich (2011): *Changing state in the plaid language*. In Cristina Videira Lopes & Kathleen Fisher, editors: Companion to OOPSLA'11, ACM, pp. 37–38, doi:10.1145/2048147.2048166.

[27] Aaron Joseph Turon & Claudio V. Russo (2011): *Scalable join patterns*. In: Proceedings of OOPSLA'93, ACM, pp. 575–594, doi:10.1145/2048066.2048111.

[28] Roger Wolff, Ronald Garcia, Éric Tanter & Jonathan Aldrich (2011): *Gradual Typestate*. In: Proceedings of ECOOP'11, LNCS 6813, Springer, pp. 459–483, doi:10.1007/978-3-642-22655-7_22.