jcc: Integrating Timed Default Concurrent Constraint Programming into JAVA

Vijay Saraswat[1], Radha Jagadeesan[⋆2], and Vineet Gupta[3]

[1] CSE Department, Penn State University, University Park, Pa 16802
[2] School of CTI, De Paul University, Chicago Il 60604
[3] Google, Inc, Mt View, Ca 94043.

Abstract. This paper describes jcc, an integration of the timed default concurrent constraint programming framework [16] (Timed Default cc) into JAVA [7]. jcc is intended for use in education and research, for the programming of embedded reactive systems, for parallel/distributed simulation and modelling (particularly for space, robotics and systems biology applications), and to support the development of constraint-based program analysis and type-checking tools.
In fully implementing the Timed Default cc framework, jcc supports the notion of (typed) logical variables (called "promises", after [5]), allows the programmer to add his/her own constraint system (an implementation of the Herbrand constraint system is provided), implements (instantaneous) defaults via backtracking, implements a complete renewal of the constraint-store at each time instant, and implements bounded-time execution of the Timed cc control constructs.
jcc implements the notion of *reactive vats* [5] as single threads of execution within the JVM; a vat may be thought of as encapsulating a single synchronous, reactive Timed cc computation. A computation typically consists of a dynamically changing collection of interacting vats (some of which could potentially be located at different JVMs), with dynamically changing connectivity.
jcc programs fully inter-operate with JAVA programs, and compile into standard JVM byte-code. jcc programs fully respect the JAVA type system; logical variables are typed. jcc is compatible with the Generic Java [3] extensions, thereby allowing the use of parameterized types. Indeed, jcc may be viewed as an extension of JAVA which replaces JAVA's notoriously difficult imperative thread-based concurrency with the notion of reactive vats interacting via constraints on logical variables.
jcc source code is available under the Lesser GNU licence through SourceForge.[4]

1 Introduction and Overview

While JAVA [7] has been remarkably successful as a new programming language, its treatment of concurrency remains extremely problematic from a conceptual point of view, and extremely difficult from a practical point of view. To a fairly traditional core of strongly-typed class-based object-oriented programming, JAVA adds threads and synchronization based on locking mutable objects in a heap shared between all the threads,

⋆ Research supported in part by NSF CCR 0244901
[4] The authors wish to thank Daniel Burrows, Lav Rai, Avanti Nadgir and Guilin Chen for their feedback and contributions to the jcc system.

together with rules that govern the flow of information between thread-specific data-structures (stack, heap cache) and the shared store [7, Chapter 17]. Surprisingly, the requirement that some seemingly natural properties should hold (e.g. coherence) leads to significant performance penalties [13]. Worse, some intuitively obvious properties (e.g. the thread-safety of immutable objects such as `String`s, double-checked locking [4] etc) do not hold. The originators of JAVA have commissioned a working group (JSR 133[5]) to fix these problems. While this work is still in progress (e.g. see [10]), clearly the development of alternate models of concurrency for JAVA seems called for.

A particularly insidious problem in reasoning about JAVA programs is that every line of JAVA code, *particularly* code not using any explicit concurrency or synchronization construct, could be executed in parallel by multiple threads on the same object, leading to potentially serious synchronization problems. That is, concurrent execution in JAVA is the default. This makes reasoning about the correctness of any piece of JAVA code extremely difficult – there is no clear separation between "sequential" JAVA and multi-threaded JAVA. One must explicitly consider all possible interleavings of method calls on an object in order to guarantee safety.

In this paper we propose jcc, a new concurrency model for loosely-coupled concurrent programming in JAVA based on the ideas of synchronous reactive programming [2, 16] (sometimes called "event-loop concurrency"). The model and the implementation are intended to support applications in education and research, for the programming of embedded reactive systems, for parallel/distributed simulation and modelling (particularly for space and systems biology applications), and to support the development of constraint-based program analysis and type-checking tools. In future work we expect to extend jcc to the richer setting of Hybrid cc (supporting a notion of continuous change, [8]), thereby opening up many more areas of application, particularly in the realm of modelling and simulation of physical, engineered and biological systems.

Vats and Ports. Fundamentally, we propose that *both* the stack and the heap (the location for objects) are private to a thread, so that *by default* mutable objects do not need to be synchronized since they can be operated on only by a single thread. Following [5], we call such a self-contained thread of execution a *vat*. (Vats are closely related to the new JAVA notion of *isolates* [19]; one may think of vats as single-threaded isolates.) A single Java Virtual Machine (JVM) instance may now be thought of as consisting of multiple vats together with a shared heap of *immutable objects*.[6] Vats communicate with each other through a few, shared, mutable "gatekeeper" objects called *ports*. Each port is located at a vat and may be "read" only by (code running in) that vat; it may be written into by any vat referencing a related immutable object called the *teller* for the port. Objects written into a port are (deep) copied from the source vat to the target vat, with the copying bottoming out on immutable objects (e.g. tellers). Ports are considered

[5] See `http://jcp.org/en/jsr/detail?id=133`, particularly the discussion of how the current specification is broken "Unfortunately the current specification has been found to be hard to understand and has subtle, often unintended, implications.... Several important issues [...] simply aren't discussed in the existing specification."

[6] An object is said to be immutable in JAVA if all its fields are `final`: that is, they must be assigned at the time of object creation and never again thereafter.

primitive objects in jcc; they cannot be written in jcc, though they can be subclassed by jcc code (cf `Thread` in JAVA).[7] Thus a port represents a single-reader, multiple-writer queue of objects. We say that a vat A *sends a message* to vat B if it writes an object into a port of B; jcc guarantees, using ideas from *wait-free synchronization* [9], that sending vats are never blocked. The *environment* $e(A)$ of a vat A may be defined as the union of the set $f(A)$ of vats that posses tellers for ports in A and the converse set $t(A)$ of vats for whose ports A possesses tellers.[8]

With such an architecture, it is natural to require that a vat A process a single message from its environment at a time, and process it to completion before receiving the next message. In processing a message, the vat may create new objects (in its local heap), invoke methods on the objects in its heap, and send messages to vats in $t(A)$. None of these operations may block; therefore if each vat may be guaranteed to execute a bounded number of operations in response to each input message, it is guaranteed to complete the processing of each input message in a bounded amount of time.

Time. We now make the observation that the receipt of a message from the world, and the computation of a response, imposes a total order on the execution of the vat. This total order is called *time* (cf. Berry's synchrony hypothesis, [12]). Note that the notion of time here is *logical* as opposed to physical – the "next" message is not required to occur at the next "second" or "millisecond". However, a particular implementation (e.g. on a real-time operating system) may guarantee that a message arrives at a vat at every chosen physical time instant, e.g. millisecond. (For such programs it must be guaranteed that the vat is able to respond, with the available compute cycles, before the arrival of the next message.) Thus the techniques in this paper can be used to program physical real-time systems as well.

The explicit introduction of time allows us to introduce a host of control structures that make it possible to describe behaviors *across time*, as we now discuss.

In JAVA-like languages, one may think of each object o as participating in an *object flow*. The elements of this flow are the objects which are communicated as arguments in message invocations on o, or returned as the result of a method invocation by o, or created by o. Immutable objects can only redirect the flow to other objects. Mutable objects can sample the flow, record it in their finite state (their set of fields), and predicate their responses to subsequent flows on this memory (= record of past state).

JAVA-like languages only allow objects to store *data* from past interactions. The methods defined on an object may be thought of as specifying the *instantaneous response* of an object to a stimulus (= method invocation) from its environment. All and only the code in the body of the method (and the code it calls, transitively) is executed on the presentation of this stimulus.

We now introduce the idea of allowing objects to store *programs* (or *agents*) from past interactions. That is, we allow an object to specify, based on current interactions, code that should be executed in the *future* (to compute the response of this object to future stimuli). To enable this, we introduce the notion of *time-based control constructs*.

[7] Subclassing may specify, for instance, how received messages are ordered before being dequeued.

[8] Because we allow tellers to be communicated in messages, these sets vary dynamically.

The statement `next {S}` is executed at the current time instant and records that the statement `S` should be executed at the next time instant (that is, the next time that the enclosing vat processes a message from its environment). Thus actions at any given instant depend not only on the particular method invocation at that time instant, but also on such agents from the past. Since there may be several such agents, and they may have been independently generated (and at different instants in the past) it becomes very convenient for us to think of them as executing (logically) in *parallel*. This is referred to as *intra-vat logical parallelism*.

How should these agents be organized so that the result of executing them is independent of their order of execution? Here we may appeal to the theory of *determinate concurrency* as developed in concurrent constraint programming [14, 16]. We view these agents as concurrent constraint programs that interact with each other by imposing and checking constraints on shared (logical) variables. (We allow both *positive* and *negative* forms of checking or asking. A positive ask suspends until a particular item is entailed by the store; a negative ask can be fired if it can be established that a particular item will never be entailed by the store for the duration of this time instant. This may require lookahead or backtracking.) We can make these agents sensitive to the data generated in the current interaction by posting that data in the constraint store at the beginning of the interaction. So as to allow the programmer to precisely define the temporal extent of items in the constraint store, we adopt the rule that by default all items in the store are dropped at the end of a time instant; therefore at the start of the next time instant only those items will be available which occur explicitly within the scope of `next` (or which are added by the environment at the beginning of that step).

Timed Default CC. Concretely, jcc may be thought of as arising from the integration of the Timed Default cc framework into (Sequential) JAVA. The Timed Default cc framework extends the CCP framework with a notion of *defaults* and *time*. To the *tell*, *(positive) ask*, concurrency as conjunction and hiding as existential idea of CCP, defaults add the idea of *negative asks*: agents may be specified that fire based on the *absence* of information (for the duration of the computation).[9] Time introduces the idea of *phased execution*: a time instant is identified with the receipt of a stimulus from the environment and the execution of a default CCP (Default cc) program to determine *both* the instantaneous response and the computation to be executed at the next time instant.

While time and defaults are conceptually completely orthogonal (i.e. the semantics of one is defined independently of the other); past work has shown that the extensions are particularly synergistic. In particular, defaults allow various *instantaneous pre-emption* control constructs (such as `do..watching`) to be expressed in the language. Indeed defaults may be used to express arbitrarily sophisticated patterns of interruption, resumption and evolution across time.

Desiderata for jcc One of our goals is to introduce Timed Default cc in the mainstream of programming practice. One pathway to this goal is the design of a completely new

[9] It is necessary to insist that negative information be stable so as to retain the notion that the result of the computation should be independent of scheduler delays or the order of execution of agents.

programming system (c.f. Oz [18]) organized around constraints and communication. However, our experience in leading several engineering teams in industry that have designed, implemented and released commercial products and services in the Internet space points to the enormous value of integrating these ideas into existing languages and environments such as C++, JAVA, and C#.

We thus set up the following design goals for jcc:

1. jcc programs should completely inter-operate with JAVA: they should be able to use JAVA class libraries (that do not explicitly use threads), and be callable from JAVA class libraries.
2. jcc programs should be strongly typed, and the type system should inter-operate with the JAVA type system, including the Generic Java extension [3].
3. jcc programs should compile to standard JVM byte codes.
4. A simple API should be provided to allow the programmer to add new constraint systems.
5. To support reflective meta-programming, jcc should provide an abstraction type for agents, and classes for each of the additional built-in control constructs for agents (e.g. `Always`, `Next` etc).
6. The implementation should be usable for small to medium-sized programs.

1.1 Comparison with Other Work

The conceptual framework of vats and promises has been borrowed from the programming language E [5], which itself has been influenced by a long line of concurrent logic programming languages. Unlike the designers of E, we have sought to realize these ideas incrementally within JAVA rather than create a new language from whole cloth.

The treatment of defaults and temporal control constructs in Timed Default cc is closely related to ESTEREL. Just as for ESTEREL, several compilation techniques for defaults and the temporal constructs are possible (e.g. compilation to finite state machines, compilation to BDDs or circuits etc). In the current version of the jcc system (described in this paper) we have chosen to implement the control constructs directly. Vats may be thought of as quite similar to the Communicating Reactive Processes of [1]. They differ in that we use a very simple form of concurrent constraint programming (ports) for inter-vat communication, instead of using the framework of CSP. This fragment is quite similar to the asynchronous π-calculus and permits dynamic collection of processes, and dynamically changing connectivity between processes.

The JAVA community has recently introduced the notion of *isolates* to enable multiple independent computations to run within a JVM [19]. Vats may be thought of as *singly-threaded* isolates that communicate declaratively with each other (via ports).

Rest of This Paper. In the next section we discuss the core language design in more detail. The design is presented in two phases. The first phase introduces vats and ports. This level is analogous to the design of a concurrent logic programming language such as Janus [17] (albeit in an object-oriented context). This language is completely usable in its own right. The second phase introduces the notion of promises, time and defaults. Next we present several simple examples of jcc programs.

The jcc system has been produced under the LGPL open source code licence. The current system, version 0.2, is available for download from SourceForge. All the features discussed in this paper have been completely implemented, except for front-end syntax processing. Instead programmers today have to use the "agent-based syntax" described below. The implementation has been used in two graduate courses, in which several hundred line long jcc programs have been developed (e.g. variations on the ESTEREL Reflex and Wrist-watch programs).

The current implementation is a few thousand lines long.

2 Language design

Syntactically, the language is obtained from JAVA by adding the constructs given in Table 1.[10] jcc may be thought of via the "equation":

$$\text{jcc} = \text{JAVA} - \text{Threads} + \text{Vats} + \text{Promises} + \text{Agents}$$

2.1 Vats

A vat is a unit of concurrent execution with its own local stack and heap. (It is associated with a JAVA thread in the current implementation.) Vats function very much like *containers* for components known as `Agents`, the analogous notion in jcc to JAVA's Enterprise Java Beans (EJBs).

Vats may be thought of as executing `Agents` and communicate with other vats through `Ports`.

Agents A vat may be created with an instance of `Actor` or `ActiveAgent`.[11] The class `Actor` extends `Port` and implements `Runnable`, and may be thought of as a component that is accessible from the vat's context (through the port) and that can be executed by the vat (very much like an EJB). An `ActiveAgent` extends `Port` and provides a method to return an `Agent`.

`Agent` is the key meta-abstraction in jcc. It objectifies an agent whose behavior extends across time. Agents allow for meta-programming: Agents are objects that can be constructed on the fly (e.g. based on incoming data), and scheduled for execution. Agents can be built from other agents. Agents allow for reactive synchronous programming within conventional JAVA syntax (that is, without the use of the control constructs described in Table 1), since built-in `Agent` classes are provided for each of the control constructs. For simplicity one may assume that every jcc computation is implemented by associating an `Agent` with a `Vat`; this `Agent` is built dynamically by executing the pure JAVA code obtained by translating jcc control constructs into invocations of constructors on the appropriate `Agent` classes. Therefore in the following we describe the

[10] Many other temporal constructs are provided, not all are listed.

[11] All the classes mentioned in this section live in the package `jcc.lang`. We use the annotation `/*filled*/` to indicate that a variable must have a non-null value. One may use an extended static checker for JAVA (e.g. [6]) to check these annotations at compile time.

jcc *contains* JAVA *less threads.* All syntactic constructs in JAVA (1.3) are permitted in jcc (including inner classes), except as indicated here. jcc programs may not use the classes `Thread` or `ThreadGroup` (from the `java.lang` package) or the `synchronized` and `volatile` keywords from JAVA.

`realized` *method keyword.* Methods defined on subclasses of `Promise` with return type `void` may be annotated with the keyword "`realized`". Such a method invocation on an object o suspends until o is realized. If the method has a return type of (a subclass of) `Promise` an unbound variable of that type is returned; this variable will be equated to the result of the actual method call made when o is realized

Additional control constructs. jcc supports the following control constructs. Below, let `p` range over promises, and `S` over *jump-closed* statements, that is, statements which are such that any jumps from within `S` (for instance, occurrences of `break`, `continue` or `return`) are directed at locations within `S`. Any variables occurring in `S` that are bound outside `S` should also be declared `final`.

`when (p) {S}`	*Run* S *once* p *is realized.*
`next {S}`	*Run* S *in the next instant.*
`always {S}`	*Run* S *at every instant.*
`every (p) {S}`	*Run* S *at every instant* p *is realized.*
`whenever (p) {S}`	*Run* S *at the first instant* p *is realized.*
`unless (p) {S}`	*Run* S *once it can be established that* p *cannot be realized.*
`watching (p) {S}`	*Run* S*, aborting it at the instant in which* p *is realized.*
`...`	
`effect {S}`	*Run* S *at the end of the current time instant.*

`jcc.lang` *classes.* User code may use the classes in Table 2, Table 3 as well as the class `Abort`:

```
public class Abort extends Exception {
    public Abort() { ...}
    public Abort( final Exception z ) {...}
    public Abort( final String z ) {...}}
```

Table 1. Syntactic Additions in jcc

```
abstract public class Promise implements Backtrackable {
    // Call with false to create a new unbound variable.
    public Promise(boolean realized){...}
    public final Promise /*filled This*/ dereference(){...}
    public final void equate(/*filled This*/ Promise other){
       ...}
    public boolean known( ) {...} // Is this realized?
    public void ensureKnown() throws Abort {...}
    public void runWhenRealized(/*realized*/ Now call)
        throws Abort {...}
    public void abortWhenRealized( ) throws Abort {...}
    public boolean equals(/*filled This*/ Object o ) {...}
    public int hashCode() {...}
    public void print(/* filled */ PrintStream o) {...}
    //Subclasses define how to equate two realized promises.
    abstract protected
     void equateBothDerefedAndRealized(/*filled*/ Promise o)
     throws Abort;}
public class Atom extends Promise {
    public final static Atom NIL = new Atom( null);
    /** Create an unbound Atom. */
    public Atom() {...}
    /** Create an atom that contains the object o. */
    public Atom( Object o) {...}
    /** Return the value associated with this atom. */
    public Object getValue() {...}
    protected void equateBothDerefedAndRealized ...}
public class Integer  extends Atom {
    public Integer() {...}
    public Integer(int o) {...}
    public Integer(/*filled*/ java.lang.Integer o) {...}
    public int intValueDeref() {...}
    public int intValue() {...}
    // this= a + b. Suspend until any two are realized.
    public void plus(/*filled*/ Integer a,
                     /*filled*/ Integer b){...}
   ... // Similarly for times.}
public class List extends Promise {
    final public Promise head;
    final public List /*This*/ rest;
    public static final List NULL = new List( null, null);
    /** Return a new promise for a list. */
    public List() {...}
    /** Return a new realized list. */
    public List(Promise head, List rest ) {...}
    public boolean isNull() {...}
    protected void equateBothDerefedAndRealized ...}
```

Table 2. Basic Promises in jcc

behavior how a `Vat` executes its associated `Agent`, and the behavior of these built-in `Agent` classes; this suffices to provide a description of how a `Vat` executes an `Actor`.

An `Agent` has three methods, all of which may be implemented by instantaneous code (that is, JAVA code not containing any of the jcc control constructs of Table 1). It is the responsibility of the vat (discussed in more detail below) to invoke these methods in a manner consistent with interpreting the code as a specification of a Timed Default cc agent whose behavior extends across time.

The vat invokes the method `now()` in order to execute the code associated with the agent for the current time instant. Agents support logical concurrency. Through the notion of *promises* and *watchers* (discussed further below), it is possible for pieces of computation in a vat to be suspended until the associated promise is *realized*. Therefore a vat also has a *scheduler* responsible for scheduling these pieces of computation. jcc makes no guarantees of the scheduler, other than a watcher will eventually be executed (within the current time instant) if the variable it is watching is realized. jcc does guarantee *single-threaded* execution for watchers: agents may assume they have exclusive access to all objects they reference (other than ports). No other thread may be modifying or accessing these objects at the same time.

Execution of this code may involve backtracking in order to resolve defaults. During this phase, the agent should not attempt to invoke any side-effects, since the code may be backtracked over, and these side-effects will not be undone. Once the agent has quiesced – note that an agent may have several concurrent sub-agents; an agent is considered to have quiesced only if all subagents have quiesced – the vat will invoke `effect()` on it to give it an opportunity to execute any side-effects (e.g. writing to various streams, sending messages to other vats).

The notion of effects is reflected in jcc syntax through the `effect` control construct. When such a control construct is encountered during execution of code by the vat, the construct is added to a list of effects (and not executed). Additions to this list are undone on backtracking. Once backtracking is complete and computation has quiesced, the list of effects is examined and executed.

Ports Vats communicate with the outside world through a collection of *ports* created by the code running inside the vat. The port is said to be located at (or owned by) the vat. A `Port` maintains an internal buffer for messages received from the *tellers* of the port; these messages may be read through the *promise* for the port. A *teller* to a port is an object that possesses the ability to send a message to the port. Messages received on a port are buffered if the receiving vat is active; otherwise the vat is activated with the port and receives the message by performing a `get` operation on the port.

Vat Life-Cycle We now describe the life-cycle of a vat. The vat executes in an infinite loop. At the top of the loop it suspends, waiting for a message on any one of its ports. The receipt of such a message triggers an "instantaneous interaction" with the code running in the vat. The message is equated with the promise associated with the port, and the `now` method associated with the current agent is executed. Once this terminates, the `effect` method is executed. Once this terminates, the `next` method is executed to determine the agent to be executed at the next time instant, and a counter tracking the time instant (as an integer) is incremented. The vat now returns to the top of the loop.

If the store becomes inconsistent at any time instant, or the agent throws an `Abort` exception, the vat terminates its execution abruptly, after *poisoning* all of its ports. The poison ultimately propagates to all the tellers of these ports, causing local exceptions to arise whenever an attempt is made to send a message through a poisoned teller. However, a poisoned vat does not automatically poison other vats; each vat has a separate constraint store.

2.2 Promises

The class `Promise` plays the same role in jcc as the class `java.lang.Object` plays in JAVA. The class is intended to be the base class subclassed by programmers to define new data-types.

A promise is a *typed logical variable*. As far as users of promise are concerned, a promise may be in one of four states: *realized*, *bound*, *unrealized and watched*, or *unrealized and unwatched*.

Typically, instances are created at subtypes of `Promise`. They may be created either as *variables* (using the nullary constructor), or as (top-level) *constants* (using any other constructor). A (top-level) constant is an instance which is not a variable but which has components (fields) that may be variable; such an object is also said to be *realized*. A variable `o` is an instance that has no data associated with it. Rather, its state changes as a result of invocations of the method `o.equate(p)`, where `p` is another promise (`o` is then said to be *bound* to `p`). Invoking the method `o.equate(p)` corresponds to posting the constraint `o=p` to the store.

If an object is neither realized nor bound, it is said to be *unrealized*.

Note that it is possible to equate a constant to another. If the two constants are different, one has a contradiction. This is handled by throwing a `FailureError`. With this version of jcc there is no support for recovering from this error. The vat is said to be *poisoned*, and a new vat must be started with a fresh agent. Poisoned vats cease to process input. Attempts to send a message to the vat through a teller raise an exception which may be caught by the teller code.

In effect, equatings represent equations imposed on the concerned variables, and the above process describes a simple *unification* algorithm (with suspension).

Suspension of Computation We now discuss two fundamental properties of promises.

First, computations may suspend on a promise until the promise is realized. This is accomplished by a new control construct in jcc. If `S` is a statement and `p` is a promise, then jcc admits the statement `when (p) do {S}`.

Such a control construct is defined as follows. If `p` is realized, `S` is executed immediately. Otherwise, `S` is *suspended* on `p`; `p` is now said to be *unrealized and watched*, and `S` is said to be a watcher for `p`. (A promise may have multiple watchers.) A subsequent invocation of `p.equate(q)` will cause `S` to be scheduled for execution if `q` is realized. If `q` is unrealized, then its watchers, if any, are merged with the watchers (if any) for `p`, and one of `p` and `q` is bound to the other. (Thus they have the same set of watchers.) If a promise is not realized, bound or watched, it is said to be *unrealized and unwatched*. (Such a promise corresponds to an unconstrained logical variable.)

Thus, as a result of `equate` method invocations (called equatings), a promise may be bound to another promise, which may be bound to another one, and so on. The *dereferenced value* of a promise is the promise that lies at the end of this binding chain. This promise may be either realized or unrealized (it may not be bound).

Automatic Dereferencing. The second important property of promises is that method invocation respects promise equatings. Methods invoked on promise `p` are *forwarded* down the chain of equatings: first the promise `p` is dereferenced to the promise `q` at the end of the chain, and then the method is invoked on `q`. Thus, any references to `this` in the code for the method on `q` refer to `q` and not `p`. By uniformly dereferencing promises before invoking methods, we maintain the invariant that the holder of a promise cannot distinguish between the promise and the value realizing the promise. This is central to the idea that promises are first-class values in jcc: a method may accept a promise as an argument, store it in a data-structure, read a value from it, place in it a value that has been separately computed.

The `realized` *Keyword on Methods* jcc adds the `realized` keyword for methods of subclasses of `Promise`. The method must be `void` or must return a `Promise`. An invocation of such a method on an object returns immediately if `o` is unrealized, suspending the body of the method on `o`. In case the method returns a `Promise` a new variable of the type of `Promise` is returned. If `o` is realized, then the body is executed immediately and the value returned.

2.3 Pre-specified agents

Table 3 enumerates the constructors for the prespecified agents. These classes objectify the Timed Default cc combinators: their constructors take arbitrary agents as arguments (and return agents). The code for these classes contains the core default and time-dependent implementation of jcc.

3 Programming in jcc

All the idioms for programming in Timed cc [15] are available in jcc. The publicly available download has several Timed cc programs.

One may also program in jcc just as one would in JAVA – using standard JAVA idioms (classes, inheritance, state, assignment, multiple methods) for system modelling. However, for public methods one should use `Promises` as arguments and as return values. This allows the computation to be structured using data-flow synchronization rather than control-flow synchronization.

Fundamentally, promises are used to model *logical concurrency*. Often it is desired to perform a certain computation on an object o (e.g. invoke a certain operation), but the object is not yet in a state in which this operation can be performed successfully. Therefore it is desired to wait until such time as some other source of change (e.g. another stimulus from the outside world) causes the object to arrive in a state in which the previous operation can be completed successfully. Promises allow such computations to

```
public class Always extends BasicAgent {
  // Run a at every time instant.
  public Always(/*filled*/ Agent a){...}}
public class ElseNext extends BasicAgent {
  // Run a at the next instant unless p is realized now.
  public ElseNext(/*filled*/ Promise p, /*filled*/ Agent a){
  ...}}
public class Every extends BasicAgent {
  // Run a at every instant in which p is realized.
  public Every(/*filled*/ Promise p, /*filled*/ Agent a){
  ...}}
public class Next extends BasicAgent {
  // Run a at the next instant.
  public Next(/*filled*/ Agent a){...}}
public class Par extends BasicAgent {
  // Run each of the argument agents in parallel.
  public Par(/*filled*/ Agent[] a){...}
  // and other similar constructors, for i=2...10.}
public class Send extends BasicAgent {
  // Send this promise on this teller.
  public Send(/*filled*/ Teller t, /*filled*/ Promise p){
  ...}}
public class Tell extends BasicAgent {
  // Equate the two promises now.
  public Tell(/*filled*/ Promise p, /*filled*/ Promise q){
  ...}}
public class Unless extends BasicAgent {
  // Run a unless p holds in the current time instant.
  // May backtrack.
  public Unless(/*filled*/ Promise p, /*filled*/ Agent a){
  ...}}
public class When extends BasicAgent {
  // Run a if p is realized now.
  public When(/*filled*/ Promise p, /*filled*/ Agent a){
  ...}
  // Run a if p is equated to q now.
  public When(/*filled*/ Promise p, /*filled*/ Promise q,
              /*filled*/ Agent a){...}}
public class WhenEver extends BasicAgent {
  // Run a at the first instant in which p is realized.
  public WhenEver(/*filled*/ Promise p, /*filled*/ Agent a){
  ...}}
public class Watching extends BasicAgent {
  // Run a; abort at the first instant when p is realized
  public Watching(/*filled*/ Agent a, /*filled*/ Promise p){
  ...}}
```

Table 3. Public constructors for pre-specified agents

be expressed directly. The first method may return immediately with a promise for the result. The invoking context may continue execution with the returned promise, blocking (using `when` or `whenever` as appropriate) only as and when it needs the returned value. In the meantime, o has recorded a computation to be performed in the future to complete this request. Thus promises allow the programmer to structure the computation in such a way that small pieces of code may remain suspended on certain events happening (certain promises becoming realized) and may produce certain other events (cause other promises to become realized). Indeed, the entire computation can be structured as lots of such small pieces – tens of thousands of such pieces. The correctness of each of these pieces is usually much simpler to check.

The semantics of ports and promises (as opposed to shared mutable objects) make them much more suitable for distributed concurrency. Here one may think of a vat running on one machine communicating with a vat running on another, through ports. This allows the two vats to continue normal operations, with a place-holder for the result, without having to suspend waiting for a response from the other.

A Bank Account Example. Consider for example a bank account. One may wish it to be programmed in such a way that it accepts method invocations to withdraw and deposit money. In both cases it should return confirmations. However, if there is not enough money to cover a withdrawal, then the withdrawal should be repeated after each of the next n deposits, and if it is still not possible to withdraw then a negative confirmation should be sent. The typical way to do this in JAVA is to synchronize on some lock object and implement the conditional wait by using `wait/notifyAlls`. Instead, we may use `Promises` as the arguments and return values. The caller of the deposit/withdraw method may proceed, leaving behind code that waits for the result to be realized. The withdraw method will realize its result at some future indeterminate point in time only when it has succeeded, or failed to do so after n tries[12].

```
public class BankAccount {
    int balance = 0;
    Boolean balanceUpdated = new Boolean();
    List pendingW = new ArrayList();
    public BankAccount() {
        every (balanceUpdated) {
          effect {
            if (! pendingW.isEmpty()) {
              for (ListIterator e = pendingW.listIterator();
                                    e.hasNext();) {
                if (((Withdrawal) e.next()).tryWithdrawal())
                e.remove();
              }}}}}
    private class Withdrawal(int amount, Confirmation c) {
```

[12] In this program, `Boolean` and `List` are from the `jcc.lang` package; for brevity we have omitted the `import` declarations. jcc supports Pizza style [11] implicit constructor declarations via syntax such as `class Shifter(int i) extends ...`. Such syntax is assumed to define `int i` as a private field in the class `Shifter`, and also define a constructor `Shifter(int i)` which initializes the field.

```
    public int count = 10;
    public boolean tryWithdrawal() {
        if (amount <= balance) {
                balance -= amount;
                confirmation.equate(new Success(genId()));
                return true;
            }
        if (count <= 0) {
            confirmation.equate(new Failure(genId()));
            return true;
        }
        count--;
        return false;
    }}
int id;
private int genId() {
    return id++;
}
public Confirmation deposit(Integer amount) {
  Confirmation result = new Confirmation();
    when (amount) {
        balance +=amount.intValue();
        balanceUpdated.equate(true);
        confirm.equate(new Success(genId()));
    }
    return result;
}
public Confirmation withdraw (final Integer amount) {
  Confirmation result = new Confirmation();
    when (amount) {
        int value = amount.intValue();
        if (balance <= value) {
            balance -= value;
            result.equate(new Success(genId()));
        } else {
            pendingW.add(
                   new Withdrawal(value, result));
        }
    }
    return result;
}}
```

Thus one can think of jcc as a general purpose language, like JAVA, differing from it only in its treatment of concurrency. Of course for modelling physical systems further restrictions may be placed, e.g. disallowing mutable state.

Conclusion We have presented the design of the language jcc which extends JAVA by replacing its concurrency model, based on ideas from synchronous reactive programming. A number of new time-based control constructs are added. The basic operations of Timed Default cc are supported.

References

1. G. Berry, S. Ramesh, and R.K. Shyamsundar. Communicating reactive processes. In *Proceedings of the 20th ACM Symposium on Principles of Programming Languages (POPL'93), Charleston, South Carolina*. ACM Press, New York (NY), USA, 1993.
2. Gerard Berry. The Esterel v5 Language Primer Version 5.21 Release 2.0. Technical report, INRIA, April 1999.
3. Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding Genericity to the Java Programing Language. In *OOPSLA*, 1998.
4. David Bacon et al. "The double checked locking is broken" declaration. http://www.cs.umd.edu/ pugh/java/memoryModel/DoubleCheckedLocking.html, 2000.
5. Mark S. Miller et al. E: Open source distributed capabilities, 1998. http://www.erights.org.
6. C. Flanagan, K. Leino, M. Lillibridge, C. Nelson, J. Saxe, and R. Stata. Extended static checking for Java, 2002.
7. James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison Wesley, 2000.
8. Vineet Gupta, Radha Jagadeesan, and Vijay Saraswat. Computing with continuous change. *Science of Computer Programming*, 30(1–2):3–49, January 1998.
9. Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.
10. Jeremy Manson and William Pugh. A new approach to the semantics of Multithreaded Java. http://www.cs.umd.edu/ pugh/java/memoryModel/, January 2003.
11. M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL'97), Paris, France*, pages 146–159. ACM Press, New York (NY), USA, 1997.
12. Gordon Plotkin, Colin Stirling, and Mads Tote, editors. *Proof, Language and Interaction: Essays in Honour of Robin Milner*, chapter The Foundations of Esterel, pages 425–454. Foundations of Computing. MIT Press, 2000.
13. William Pugh. The JAVA memory model is fatally flawed. *Concurrency:Practice and Experience*, 12(1):1–11, 2000.
14. Vijay Saraswat. *Concurrent Constraint Programming*. Doctoral Dissertation Award and Logic Programming. MIT Press, 1993.
15. Vijay Saraswat, Radha Jagadeesan, and Vineet Gupta. Programming in timed concurrent constraint languages. In B. Mayoh, E. Tyugu, and J. Penjaam, editors, *Constraint Programming: Proceedings 1993 NATO ASI Parnu, Estonia*, pages 361–410, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 1994. Springer Verlag.
16. Vijay Saraswat, Radha Jagadeesan, and Vineet Gupta. Timed default concurrent constraint programming. *Journal of Symbolic Computation*, 22(5–6):475–520, November–December 1996. Extended abstract appeared in the *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages*, San Francisco, January 1995.
17. Vijay Saraswat, Kenneth Kahn, and Jacob Levy. Janus: A step towards distributed constraint programming. In *North American Logic Programming Conference*, October 1990.
18. Gert Smolka. The Oz programming model. In Jan van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, volume 1000 of *Lecture Notes in Computer Science*, pages 324–343. Springer-Verlag, Berlin, 1995.
19. Peter Soper. The Application Isolation API, 2001. JSR 121, http://www.jcp.org/en/jsr/detail?id=121.