EJAVA — CAUSAL EXTENSIONS FOR JAVA

Alexandre Santoro

Walter Mann

Neel Madhav

David Luckham

Technical Report: **CSL-TR-98-768**

(Program Analysis and Verification Group Report No. 79)

August 1998

This project is funded by DARPA under Air Force Rome Labs Coop-Agreement number F30602-96-2-0191 and under SRI, subcontract number C-Q0545.

eJava — Causal Extensions for Java

Alexandre Santoro Walter Mann Neel Madhav David Luckham

Technical Report: CSL-TR-98-768
Program Analysis and Verification Group Report No. 79

August 1998

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305-4055

Abstract

Programming languages like Java provide designers with a variety of classes that simplify the process of building multithreaded programs. Though useful, especially in the creation of reactive systems, multithreaded programs present challenging problems such as race conditions and synchronization issues. Validating these programs against a specification is also not trivial since Java does not clearly indicate thread interaction. These problems can be solved by modifying Java so that it produces *computations*, collections of events with both causal and temporal ordering relations defined for them. Specifically, the causal ordering is ideal for identifying thead interaction. This paper presents *eJava*, an extension to Java that is both event based and causally aware, and shows how it simplifies the process of understanding and debugging multithreaded programs.

Key Words and Phrases: eJava, Java, causal model

eJava– Causal Extensions for Java[1]

Alexandre Santoro Walter Mann Neel Madhav David Luckham

Programming languages like Java provide designers with a variety of classes that simplify the process of building multithreaded programs. Though useful, especially in the creation of reactive systems, multithreaded programs present challenging problems such as race conditions and synchronization issues. Validating these programs against a specification is also not trivial since Java does not clearly indicate thread interaction. These problems can be solved by modifying Java so that it produces *computations*, collections of events with both causal and temporal ordering relations defined for them. Specifically, the causal ordering is ideal for identifying thead interaction. This paper presents *eJava*, an extension to Java that is both event based and causally aware, and shows how it simplifies the process of understanding and debugging multithreaded programs.

1 Introduction

Programming languages like Java provide designers with a variety of classes that simplify the process of program development. Some of these classes, such as threads, allow one to easily build concurrent programs. Multithreading is especially useful in the implementation of event-based systems, where one thread monitors user events and starts other threads to handle the user's requests. This is the model behind Java's AWT[GYT96].

Though very useful for this kind of programming, multithreading creates a series of problems, such as race conditions and synchronization issues. Discussing the issues related to concurrent programming is beyond the scope of this text. For that the reader is referred to [Ari90] and [Sch97]. It is sufficient to say that understanding and debugging multithreaded programs is a complex task.

As an example, suppose that a specification of a Java program requires two threads. Thread T1 prints sequentially the letters A, B and C, while thread T2 prints the numbers 1, 2, and 3. The execution of such a program could result in an output such as: `A,1,2,3,B,C`. Printing to

[1] A previous version of this paper was published in the *Proceedings of the 10th International Conference on Software Engineering and Knowledge Engineering — SEKE '98, pages 251–260.*

the standard output, in this case enforces thread synchronization and produces a totally ordered output.

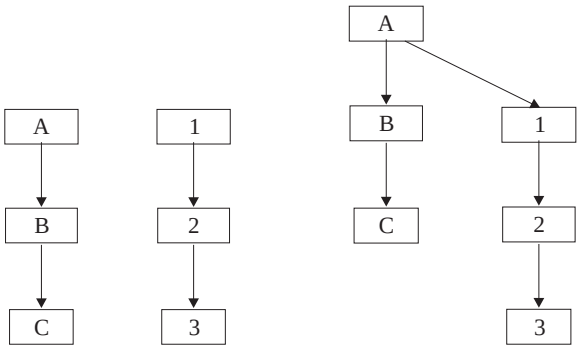
Now suppose that the specification of the program was modified, adding the restriction that T2 must only start printing after T1 has printed `A`. The output of the first program apparently satisfies this condition, though source code analysis would be necessary to verify that T2 actually waited for T1 before it started printing; it may just be a coincidence that the output is the desired one. Besides, different runs could shuffle the results differently, even if the program was executed in exactly the same way. Clearly, temporally ordered output is not sufficient for error detection in thread-based programs.

Attempts have been made in several areas to deal with the problem of validating and analyzing concurrent programs. Woods and Yang, for example, suggest the use of domain knowledge and source code analysis as a way to understand program execution[WY95]. Li and Shigo tackled the problem of understanding complex concurrent systems in the domain of SDL, a language for describing telecommunication systems[LS92]. Their approach consisted of a framework based on simulated components trying to capture the characteristics of the actual system. Finally, Tombos et al. defined event-driven reactive component semantics based on event histories in order to describe the higher level semantics of the system[TGD97]. All these approaches fail in that they base their deduction of model behavior on domain knowledge, and not just on the model itself. They are useful for defining and understanding systems, but aid only indirectly in the process of debugging and validation.

Another approach to debugging and checking programs, concurrent or not, has been by the use of formal language extensions. Two notable examples of such languages are Anna[Luc90] and Eiffel[Mey88]. Through the use of assertions in Anna, or invariants in Eiffel one is able to define the expected relation between input and output of objects, as well as the behavior of the object itself. These constructs are helpful in debugging and guaranteeing correctness, but they do not aid in understanding concurrent programs.

Rapide[LKA+95] presents an interesting solution to this problem. Rapide is an architectural prototyping language that, when executed, produces *computations*: a collection of events and two ordering relations between them. Events represent any activity of interest in the program, and the ordering relations are timing and *causality*. Timing relations describe the temporal ordering of the events; causal relations attempt to define an ordering for the events based on the "execution flow" of a program. For more details on Rapide the reader is referred to [Luc96] and [Luc98].

Computations may be viewed as directed acyclic graphs, with the nodes representing events and the arcs representing the causal relation between them. This representation makes it possible



(a) Independent (b) Interacting

Figure 1: Specification of behavior

to differentiate between program threads that interact and those which do not, as may be seen in figure 1. Figure 1(a) shows a computation with two independent threads executing without interfering with each other; figure 1(b) shows a computation where the second thread waits for the first one to print before it starts its own execution. This difference is clearly indicated by the presence of the extra arc in the second figure.

eJava is an attempt to extend Java with the same kind of functionality found in Rapide. It is an non-intrusive extension of Java that has, as a a side effect the production of event information. This information is then used by a *logger*, a tool that extracts events and their causal relation from an event data set to produce a computation. This computation is a representation of the program's execution and may then be analyzed for errors and unexpected synchronization behavior.

In order to extend Java in this way it is necessary to first build a *causal model* for the language, a set of rules that define the several causal relations between events generated by an *eJava* program. Such a causal model is an collection of rules based on a *causal model theory*, which is an abstraction of the important aspects of causal relations.

The rest of this paper presents a subset of *eJava*, concentrating on the non-distributed aspects of the language (for a more complete definition of *eJava* the reader is referred to [Man98]). It starts by presenting computations and causal theory in sections 2 and 3. Section 4 applies this theory to the definition of *eJava* and its causal model. Section 5 shows an example of an *eJava* program. Finally, section 6 summarizes our results and gives some suggestions of future work.

2 Computations, Time and Causality

Computations are a convenient representation of the execution of distributed and/or parallel systems. They consist of a set of events, $\mathcal{E}$, and two ordering relations on $\mathcal{E}$: a temporal relation $\mathcal{T}$ and a *causal* relation $\mathcal{K}$.

The temporal relation $\mathcal{T}$ orders the events in $\mathcal{E}$ with respect to one or more clocks. If a and b are events ordered by $\mathcal{T}$ we state that $a \leq b$. This relation has several interesting properties. First, it is both reflexive and transitive. Second, it provides a *total order with respect to a clock* for the events; if two events a and b share a clock, then at least one of $a \leq b$ or $b \leq a$ must be true.

The causal relation $\mathcal{K}$ defines an ordering with respect to the *cause* of an event. Cause here is distinct from the philosophical or statistical notion of cause. Rather, it is to be thought of as an ordering based on the control flow, data flow and synchronization points of a program's execution. What exactly determines this ordering is dependent on the *causal model* used for that particular language or system. Furthermore, a language or system might have more than one causal model.

Given two events a and b, if a causally precedes b then we say that $a \prec b$. This causal relation has some interesting properties. First, it is irreflexive; events cannot causally precede themselves. Second, it is assymetric; if $a \prec b$ is true, then $b \prec a$ must be false. Third, it is a transitive relation; if $a \prec b$ and $b \prec c$ then $a \prec c$. Finally, the causal relation defines a partial order; it may be the case for two events a and b that neither $a \prec b$ nor $b \prec a$ is true. In this case we state that a and b are *independent* and write it as $a \parallel b$. Thus, with respect to causality, computations may be represented as directed acyclic graphs as mentioned before. To keep the visual representation of the graph simple, often only the transitive reduction is shown.

3 Causal Modeling

A *causal model* of a language or system is a set of rules formalizing the conditions under which two events are causally related. Causal model theory is based on three concepts: threads, events, and connections. Once these concepts are defined for the system or language being studied, source-code analysis and/or execution of the system produce a set of events $\mathcal{E}$, a temporal relation $\mathcal{T}$ and a set of connections $\mathcal{C}$. $\mathcal{E}$, $\mathcal{T}$ and $\mathcal{C}$, together with the rules and theory presented in this section are all that is necessary to derive the causal relation $\mathcal{K}$ and create a computation.

The theory presented here is a simplification of a more complex model. It makes some assumptions that are pertinent only to the *eJava* case. First, it assumes that one is dealing with a single program and that the system has only one clock. Second, it assumes that there are no propagation delays to contend with. A causal modeling theory that addresses these issues is the subject of a

future paper.

The rest of this section will present a theory of causal modeling. It will first define the basic concepts of threads, events and connections. Rules are then presented for determining whether two events are causally related or not.

3.1 Definitions

Threads

Threads are the producers of events and the basic sequencing components of a system, providing a total order for the events they produce. Threads, then, are a representation of a "control flow" ordering of the system under study.

Associated with each thread is a counter which is incremented according to the causal model. Whenever a thread generates an event, the current value of its counter is passed along with it.

Events

Events are a representation of the activities of interest in the system being analyzed. What defines an activity as being of interest is system dependent and part of the process of causalmodel definition. In some cases, determining what constitutes an event requires prior knowledge of the system under study. More often, it can be derived exclusively from properties of the language used to build the system, as is the case with *eJava*.

What information an event contains is also system dependent. Different activities might require that different information be conveyed and thus the elements of an event may vary. Nevertheless, there is a minimum amount of information that an event must have in order for one to reconstruct causality. These include a name, the time of creation (according to some clock), the thread that created the event and the value of the thread's counter.

Formally, events can be defined as:

Definition 1 *An event e is a n-tuple containing at least the following elements:*

$$e = (e_n, e_t, e_p, e_v, \cdots) \tag{1}$$

where e_n is the event's name, e_t is the time of event creation, e_p is the associated thread and e_v is the value of the thread counter when the event was generated.

Event names may be composed of many parts and convey most of the information necessary for causal ordering. One part of the name, for example, might define a generic category of an event,

while another indicates a specific case of that category and a third part could indicate a unique ID for that event. For example, a system might create an event named `ReadVar.Foo.12` where `ReadVar` is a generic description of the action (reading a variable), `Foo` indicates which specific variable is being read, and `12` indicates it is the twelfth access of such a variable. The implication of this is that event names need not be known before system execution.

Connections

Static connections[2] are a way of representing ordering that is due to an exchange of information between parts of a system and thus represents the "data flow" ordering component of causality. This exchange might happen through message passing, data sharing or any other mechanism that provides a *synchronization point* between activities happening in possibly distinct threads.

Connections must have at least two elements: a source name and a destination name. Formally one would state:

Definition 2 *A static connection is an ordered pair with the following elements:*

$$c = (c_s, c_d) \tag{2}$$

where c_s is the name of the source *event and c_d is the name of the* destination *event.*

For example, suppose a Java program had a statement of the form `A = B;`. One might want to state that there is an ordering there imposed by the fact that `A` gets the value some thread last wrote to `B`. Since the thread that last wrote to `B` and the the thread executing this statement might be different, one might want to make this ordering explicit through the use of a connection. Such a connection could be written as

```
( Write.B Write.A )
```

if `Write.A` and `Write.B` were the names of the events corresponding to writing to `A` and `B` respectively.

3.2 Rules

Execution and/or analysis of the system should produce the event set $\mathcal{E}$, the temporal relation $\mathcal{T}$ and the set of connections $\mathcal{C}$. Two basic rules of causality are then applied pair-wise to the elements of $\mathcal{E}$ in order to determine their causal relation. One rule orders the events with respect to threads and the second orders them with respect to connections. These rules use $\mathcal{T}$ and $\mathcal{C}$ in order to determine whether there is a causal arc between the two events or not.

[2] *Dynamic connections* are not necessary for the definition of the *eJava* causal model, so they will not be described here.

Thread-based ordering

The first ordering in any causal model is imposed by the sequential nature of threads of execution. Once a thread is defined, events generated by that thread are sequential in nature and are thus fully ordered. The thread counter keeps track of the ordering within the thread. This relation can be formally defined by the following rule:

$$\forall\ a, b\ \epsilon\ \mathcal{E}\ \ : \{a_p = b_p, a_v < b_v\} \Rightarrow a \prec b \tag{3}$$

This rules is intended to guarantee that if a thread generates two events sequentially, incrementing its counter value between event generation, the events will be causally ordered.

Connection-based ordering

The second ordering in any causal model is imposed by the connections. If two events are part of the computation and there is a connection "joining" them, then a causal ordering is defined between them. Formally, this can be expressed as follows:

$$\forall\ a, b\ \epsilon\ \mathcal{E}, a \neq\ b : \{\exists\ c\ \epsilon\ \mathcal{C}\ \mid\ \ a_n = c_s, b_n = c_d, a_t \leq b_t\} \Rightarrow\ \ a \prec b \tag{4}$$

The conditions are stating that a connection a and b should respect the temporal ordering, and the connection's delay. It also states their names should properly match the connection's source and destination event name. Take, for example, the connection shown in section 3.1. Given the events a = `( Write.B 0 t1 4 )`, b = `( Write.A 4 t2 4 )` and c = `( Write.B 5 t3 2 )`, applying the connection ordering rule would show that $a \prec b$, but not $c \prec b$ since it does not satisfy the time ordering condition.

3.3 Connection Groups

Events are always generated during run-time execution of the system, while connections may be extracted from three different sources: the causal model for the language, analysis of the programs source code or run-time execution of the system. Depending on the causal model desired, it might not be possible to know all connections before execution. This complicates the process of building the causal relations, especially if one does not have access to the program's source code.

A solution to this problem is collecting connections into groups where event names are identified by patterns instead of explicitly. Using patterns to describe these connection groups may allow one to build causal models without having to do source code analysis.

For example, suppose the causal model requires that all reads to a variable follow the last write to that variable. Unless one knew the name of all variables beforehand, these connections could only be derived at run-time, which would require more complex processing of the event data. Even if analysis of the source code provided information about all the variables in the system, one would have to list each connection explicitly. In an *eJava* program with three variables, A, B and C one would have to create the connections:

```
( Write.A Read.A )
( Write.B Read.B )
( Write.C Read.C )
```

Clearly, this could be simplified if one could specify *connection groups*. All connections are based on name matching and since event names are strings, a way to approach this problem is by using string pattern matching in the event names. This would allow one to causally relate any events that matched the "in" and "out" patterns of a connection.

Rather than create a new pattern matching language, we borrow ideas from the pattern language defined for Tcl[Ous96]. Of interest to us are just two operators: "`*`" which will match anything, and "`\?`", where "`?`" is a number, which will match whatever the corresponding "`*`" in the previous pattern matched (i.e, 1 corresponds to the match of the first "`*`", 2 to the second, etc.).

Using this pattern language, connection groups would take the form:

(*Pattern_with_** *Pattern_with_\ ...*)

Consider the variable read/write connections mentioned above. Supposing that all "write to variable" events are of the form `Write.<varname>` and all the "read variable value events are of the form `Read.<varname>`, and that these are the only events that have names starting with `Read` and `Write`, one could express the connections as:

```
( Write.* Read.\1 )
```

which states that a `Write` to any variable should causally precede a `Read` to a variable with the same name.

This notation provides two major advantages. First, it is much shorter than listing all the possible variable read and write events in the system. Second, it enables one to define connections without the need for source code analysis.

4 *eJava* Definition

4.1 Overview

eJava is an extension of Java for the purpose of instrumenting Java programs to generate computations. All extensions to Java occur within *formal comments*, special symbols that indicate *eJava* constructs but are treated by Java compilers as simple comments (and hence ignored). Thus, every *eJava* program is a legal Java program. The execution semantics of an *eJava* program is identical to the execution semantics of the underlying Java program. Furthermore, every Java program is also a legal *eJava* program; even without explicit user generation of events, the implicitly generated events yield an overall view of program execution and thread synchronization.

The only lexical extension defined in *eJava* is the formal comment symbol, `//+`. These formal comments give the user some flexibility and control over the computation produced. It is through them that explicit events are defined and generated.

Figure 2 shows a view of an *eJava* system. In it, *eJava* code (Java code plus *eJava*'s formal comments) is processed by an *eJava* compiler to produce bytecode that is then executed in a Java virtual machine. The output of this machine is two-fold: the results of executing the program normally (as if it were a regular Java program), and a file containing event information. This file is then processed by a *logger*, which creates a computation based on the information contained in the file and the causal model for *eJava*. This computation can be used as the input to several computation-aware tools such as partial order viewers and animators.

The rest of this section defines the causal model for *eJava*, not including those constructs and definitions that relate to distributed programming, which will be the subject of another paper. Again, the reader is directed to [Man98] for a complete specification of the *eJava* language. The components required by the theory presented in section 3.1 (threads, events and connections) are defined here.

4.2 Threads

Threads in the *eJava* causal model are the same as threads in Java. A thread's name may be found by calling the `getName()` method for that class. Each thread has a counter which is incremented whenever an event is generated that has the thread as its associated thread.

4.3 Events

Events identify any activity of interest in a system. In the case of *eJava* there are two different types of events: explicit and implicit. Explicit events are user-defined events created by formal

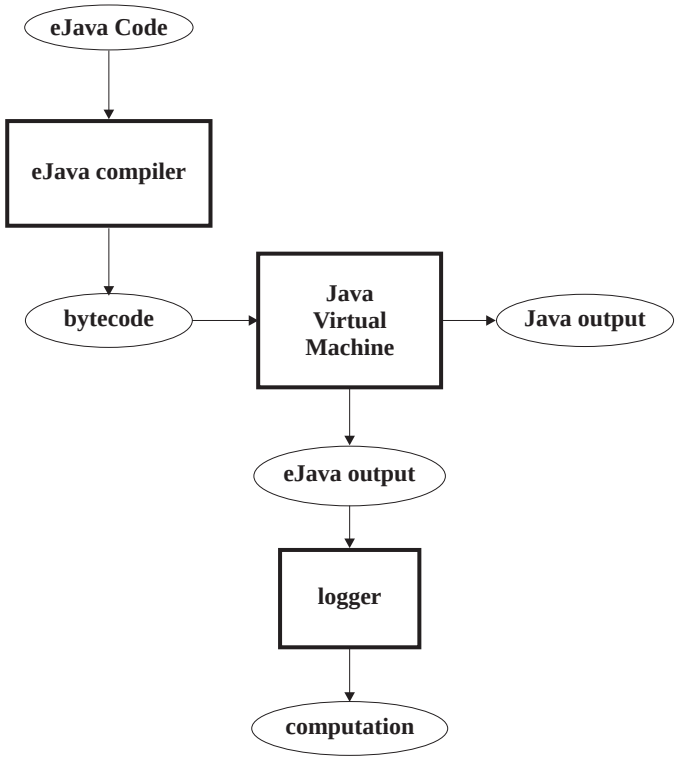


Figure 2: An *eJava* system.

annotations to a Java program, while implicit events are generated automatically.

Whether explicit or implicit, events have the same format and the same attributes. Some of these attributes are required for reconstructing causality, while others add information that might be of interest to the developer. For the purpose of building a causal model, the attributes of interest are:

- **name** The name of the event. This is used not only for identification purposes but also when applying the connection rules in in order to create a causal ordering.
- **associated thread** The thread (as given by the call to `java.lang.Thread.currentThread()`) executing the statement or declaration that caused the event to be generated. An event is considered generated by a particular thread if, at the point where the event was generated, a call to currentThread() would yield that thread.
- **time stamp** Usually a pair of values denoting when the event started and ended. In *eJava* the start time and end time are always the same, so the timestamp is an integer, the value

of `System::currentTimeMillis()` on execution of the statement that caused the event to be generated.

Of the three parameters mentioned above, two are extracted automatically from the system, leaving event names to be defined by the causal model. Proper name selection is important in that it will determine the connection rules for the causal model.

4.3.1 Explicit Events

Explicit events are generated by `//+ perform` statements, which may occur anywhere a Java statement may occur. These statements define the event name and its parameters, and when a `//+ perform` statement is executed, it causes an event to be generated. The programmer has no control over the value of the thread, counter and time parameters of the event; these are generated automatically by *eJava*.

Explicit events allow the *eJava* programmer to add more information to a computation. For instance, the events' parameters may provide additional information on the state of the program at the point in execution where the event was generated. Explicit events may also be used to indicate the occurrence of an activity in the system that would not be logged automatically by the implicit events. For example, explicit events could be used to indicate when the value of a variable becomes negative.

4.3.2 Implicit Events

Implicit events are automatically generated by the execution of an *eJava* compiled program. They are generated when certain language specific activities of interest are executed. The following is a list of these events:

- `Object_Create.<objname>`

  This event is generated at the point where the object is created. The creation of an object is defined as the point at which the constructor begins executing. `<objname>` is the name returned by a call to the object's `hashCode()` method.

- `Object_Finalize.<objname>`

  This event is generated whenever an object is finalized. Finalization is defined as the point where the finalize method for the object is called. `<objname>` is the name returned by a call to the object's `hashCode` method.

The creation and finalization of the "main" object (the static class object that the main program is part of) are not logged. Creation of objects of predefined (or library) classes are suppressed by default. They may be manually unsuppressed.

- `Variable_Read.<varname>`

  This event is generated whenever a primitive variable is actually read. `<varname>` is the name of the primitive variable.

- `Variable_Write.<varname>`

  This event is generated whenever a primitive variable is actually written. `<varname>` is the name of the primitive variable.

  Local variables (variables declared in methods) are not logged because they can never be accessed by more than one independent thread, and thus reads and writes on local variables do not introduce any causal arcs.

- `<M>_Call`

  This event is generated when the method of an object is called (i.e. when the method itself begins executing). `<M>` is the method's name.

- `<M>_Return`

  This event is generated at the completion of the return from the method (i.e. when the caller begins executing again.) `<M>` is the method's name.

- `Thread_Start_Call.<threadname>`

  This event is generated when the `start()` method of a thread object is called. `<threadname>` is the name of the thread that will be started, as given by a call to the thread's `toString()` method. It is NOT the name of the currently executing thread.

- `Thread_Start.<threadname>`

  This event is generated each time the `run()` method of a thread starts. `<threadname>` is the name of the thread.

- `Thread_End.<threadname>`

  This event is generated each time the `run()` method of a thread ends. `<threadname>` is the name of the thread.

- `Thread_Synchronize_Start.<objname>`

  This event is generated whenever thread synchronization begins (i.e. whenever a synchronized method or a synchronized statement begins executing). It is also generated right after the completion of a `wait()` statement. `<objname>` is the name of the object whose lock is acquired.

- `Thread_Synchronize_End.<objname>`

  This event is generated whenever thread synchronization ends (i.e. whenever a synchronized method returns or a synchronized statement finishes executing). It is also generated right before the execution of a `wait()` statement on an object. `<objname>` is the name of the object whose lock is released.

eJava ensures that every object, thread and primitive field variable in a program has a unique name, which is used in the event information. The name of a static class object is simply the name of the class. The name of every other object (including a thread object) is derived by concatenating the name of its class and the String image of a call to the `hashCode()` method of the object. The name of a static primitive field variable is derived by appending the name of the variable to the name of the class. The name of a primitive field variable in an object is derived by concatenating the name of the object (as defined above) and the name of the variable.

In the actual implementation of *eJava*, only the first part of the name (the one until the first '.') was used as the event's name. The rest of the name became event parameters and the logger is responsible for reconstructing the complete name in order to apply the causal model. This was done so as to improve the legibility of computations.

4.4 Connection Rules

The final step in defining the *eJava* causal model consists of listing the rules for creating connections. In the case of *eJava* connection groups are sufficient to complete the description of the causal model. The implication of this is that no source code or execution analysis is necessary to define all the connections for the system. This allows one to "hardwire" the causal model directly into the logger.

Here is the connection groups that define the connections in *eJava*:

1. `(Object_Create.* Thread_Start.\1)`

   This group establishes that the creation of a thread causally precedes the start of its execution.

2. `(Thread_Start_Call.* Thread_Start.\1)`

This group establishes that the start of the execution of a thread must causally follow the point in execution of the thread that originated the `Thread::start()` call to it.

This presents a problem. In Java, any object that has visibility to a Thread object may start it by calling its `start()` method. In particular, a thread other than the one that created the Thread object may start it. If multiple threads independently call the `start()` method of a thread, the result is undefined by the Java language. Further, starting a thread results in the call of its `run()` method, with no indication of which object or thread started the new thread. Since there is no way for an observer to tell which thread started the new thread, the `Thread_Start` event causally follows some indeterminate subset of the `Thread_Start_Call` events that satisfy the group above.

3. `(Variable_Write.* Variable_Read.\1)`

   This group just orders the variable reads with respect to the variable writes. Since primitive variables might be accessed by more than one thread, this group guarantees the causal ordering between reads and writes.

4. `(Variable_Write.* Variable_Write.\1)`

   The sequence of `Variable_Write` events denotes the order of writes to the variable. Like in the previous connection group, this is a result of the fact that more than one thread may have access to a primitive variable.

5. `(Thread_Synchronize_End.* Thread_Synchronize_Start.\1)`

   This group guarantees that any activity that happens in an area of mutual exclusion (i.e. a synchronized method call) will be totally ordered.

Note that this is only one of many possible causal models. Other models are possible with more, less, or possibly different connection rules. For example, the model above does not order the events corresponding to reading a specific variable. If it were desirable to have this ordering explicit in the model it could be achieved by adding the group:

```
(Variable_Read.* Variable_Read.\1)
```

Different causal models would allow one to interpret the same set of events in different ways and gives one the ability to explore the program space in a new way.

5 Example

5.1 Description

Suppose one wants to use *eJava* to build a system that models a fisherman and a fish. The fisherman's behavior is quite simple: he fishes until he either catches a fish or gets tired of waiting. Either way, when one of these two condition is met he goes home.

The fish is not as lucky. Its life consists of swimming until it finds some food and eats it. The problem is that the food is either the fisherman's bait or toxic waste. Whatever he eats, the next thing the fish does is die.

One way to implement this code is by having two different threads, one for the fish and one for the fisherman. During execution each thread will produce explicit events indicating its state[3]: the fisherman thread will either be "fishing" or "going home;" the fish thread will be either "swimming," "biting," or "dying."

5.2 Code

Implementing the specification in *eJava* led to the code below for the fisherman:

```
class fisherman extends Thread {
  fisherman() {}

  //+ action Fish();
  //+ action GoHome();

  static int tillBored = 100;

  public synchronized void run() {
    try {
      //+ perform Fish();
      wait(tillBored);
    } catch (InterruptedException e) { }

    //+ perform GoHome();
  }

  public synchronized void do_notify() {
```

[3]Though convenient, this is not actually necessary. The same information could be derived from the implicit events.

```
      notify();
  }
}
```

eJava added four lines to the standard Java code. The two `//+ action` declarations define the types of user-defined events that the class can produce. The two `//+ perform` statements in the body of the `run()` method, when executed, generate the events defined by the actions.

The code for the fish class is:

```
class fish extends Thread {
  //+ action Swim();
  //+ action Bite();
  //+ action Die();

  fisherman the_fisherman = null;
  boolean take_bait = false;

  fish(fisherman o, boolean bait) {
         the_fisherman = o;
         was_bait = bait;
  }

  public void run() {
    //+ perform Swim();
    //+ perform Bite();
    if (was_bait) {
      the_fisherman.do_notify();
    }
    //+ perform Die();
  }
}
```

Again, it is straightforward: `//+ action` declarations define the events the class may produce, while the `//+ perform` statements generate those events. Note that whether the fish will eat the bait or the toxic waste is determined by a parameter in its constructor just for convenience; the program might as well have used a random number generator in order to decide which action to take.

Finally, the main code for the example is:

```
public class test {
  public static final void
         main(String []args)
  {
    fisherman CaptainBob =
      new fisherman();
    CaptainBob.start();
    fish Goldie =
      new fish(CaptainBob, args.length>0);
    Goldie.start();
  }
}
```

It is straightforward Java code, with nothing special about it. It just creates instances of each class and starts them. The program terminates when all threads finish running. It also takes as a command line argument to determine whether the fish will bite the bait or the toxic waste (no arguments, toxic waste; any argument, bait).

5.3 Results

Before going into the results of executing the *eJava* program, let us discuss what would happen if it had been compiled by a Java compiler, with print statements where there currently are `//+ perform` statements. The result would be a totally ordered list of events that would not produce a complete set of information. Since the fish bites before knowing what happens, extra print statements are necessary to determine what caused the death of the fish. As for the fisherman thread, it never knows why it quit the wait process and therefore is unable to inform if there was a bite or not. Clearly, more code would be necessary to provide all this extra information.

Execution of the same program under *eJava* creates the computation in figure 3. This computation contains all the events, both the user-defined and implicit ones. It includes events for object creation, thread synchronization and variable access. The causal ordering derivation is also quite simple. The edges labeled with an `A` indicate the causal arcs that are due to connection rule 2. The arcs labeled `B` are due to connection rule 5. All other arcs are a result of applying the thread ordering rule.

Though figure 3 gives a good idea of what goes on during the execution of the *eJava* program, it goes into a little bit too much detail for ease of understanding. This problem can be easily solved by filtering the computation so that it contains only the user-defined events. This leads to the computations in figure 4. In this figure it is obvious which one is the computation in which the fish

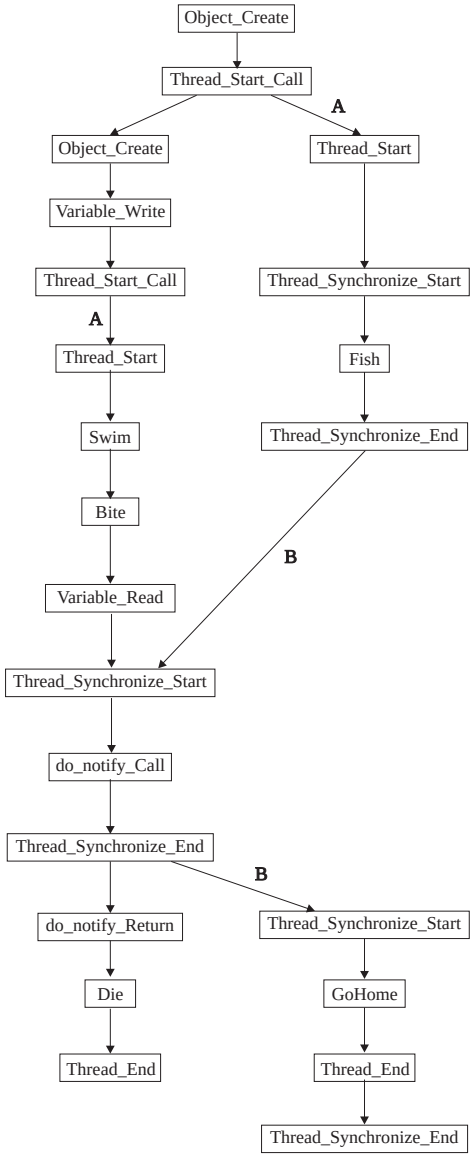
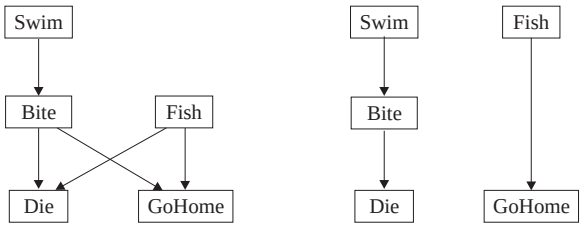


Figure 3: Complete computation for biting case.



(a) Bait (b) Toxic waste

Figure 4: Reduced computations

bites the bait and which one is the computation in which he eats the toxic waste. As can be seen, the program unambiguously defines what happened during execution, making the thread-interaction (when and if it happens) explicit.

6 Conclusion

This paper presented the definition of an extension to Java called *eJava*. Java was extended with causal semantics which, without any loss of functionality, resulted in program executions that generated computations. These computations are much more descriptive of what happens during the run of a program, which made understanding (and therefore debugging) multithreaded programs easier. Maybe the greatest benefit of computations and *eJava* is the ability to easily identify the points where threads interact, either through variable sharing or synchronization methods.

Computation-producing, causality-aware programming languages and systems provide many other benefits than easier debugging of concurrent systems. First, they allow one to build formal models of systems that are more descriptive. The partial-order nature of computations allows one to specify independent components of the system, as well as make explicit the points of interaction. The computations generated by such languages also allow one to visually identify the parallel components of a system as well as its bottlenecks. These characteristics, as presented in the Rapide language, have been used to build an executable standard for the SPARC-V9 instruction set architecture[SPL95].

One other great advantage of computations is that one can use maps to build new computations in order to validate systems[GL92][Mel90]. Maps are objects that take computations as inputs and produce computations as output. Maps look for specific event patterns in the input computation and when a match is found add events to the output computation. Parameter values from the pattern match may be used in the creation of the output events.

Maps can be used for multiple purposes. First and foremost, they allow one to create computations that are an abstraction of the input computation. For example, in the SPARC-V9 model mentioned earlier, all the events that make up one instruction could be abstracted to one event corresponding to the whole instruction. A series of maps could then be used to provide different views of the same system.

Another use of maps is that they are an easy mechanism for building automatic constraint checking tools that not only verify timing constraints, but also guarantee that independence and parallelism is respected. For an example of such an application of maps the reader is referred to [Ken96].

The authors believe that computations are a useful, information-rich way of presenting information about the behavior of distributed and concurrent systems. Current work is under way by the Program Analysis and Verification Group developing tools for producing, viewing and analyzing computations.

7 Acknowledgements

The authors would like to thank James Vera, John Kenney, and the other members of the Program Analysis and Verification Group of the Computer Systems Laboratory at Stanford University for their help and support in the development of *eJava*.

References

[Ari90] Moti Ben Ari. *Principles of Concurrent and Distributed Programming.* Prentice-Hall, 1990.

[GL92] Benoit A. Gennart and David C. Luckham. Validating discrete event simulations using event pattern mappings. In *Proceedings of the 29th Design Automation Conference (DAC)*, pages 414–419, Anaheim, CA, June 1992. IEEE Computer Society Press, **Best paper award**.

[GYT96] James Gosling, Frank Yellin, and Java Team. *The Java Application Programming Interface Volume 2: Window Tookit and Applets.* The Java Series. Addison-Wesley, 1996.

[Ken96] John J. Kenney. Executable formal models of distributed transaction systems based on event processing. Technical Report CSL–TR–96–710, Computer Systems Lab, Stanford University, November 1996.

[LKA+95] David C. Luckham, John J. Kenney, Larry M. Augustin, James Vera, Doug Bryan, and Walter Mann. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, April 1995.

[LS92] Xiaofeng Li and Osamu Shigo. A simulation-based SDL support system. In *Proceedings of the Fourth International Conference on Software Engineering and Knowledge Engineering*, pages 284–291, 1992.

[Luc90] David C. Luckham. *Programming with Specifications: An Introduction to ANNA, A Language for Specifying Ada Programs.* Texts and Monographs in Computer Science. Springer-Verlag, October, 1990.

[Luc96] David C. Luckham. Rapide: A language and toolset for simulation of distributed systems by partial orderings of events. In Doron A. Peled, Vaughan R. Pratt, and Gerard J. Holzmann, editors, *Workshop on Partial Order Methods in Verification*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pages 329–357. Princeton University, 1996.

[Luc98] David C. Luckham. Rapide: A language and toolset for causal modeling of distributed system architectures. In *Proceedings of the Second International Conference on World-Wide Computing and its Applications*, pages 88–96. Springer-Verlag, 1998.

[Man98] W. Mann. ejava: An extension of java for event generation. Technical Report CSL–TR–98–757, Computer Systems Lab, Stanford University, apr 1998.

[Mel90] Sigurd Meldal. Supporting architecture mappings in concurrent systems design. In *Proceedings of Australian Software Engineering Conference.* IREE Australia, May 1990.

[Mey88] B. Meyer. Eiffel: Reusability and Reliability. In Will Tracz, editor, *Software Reuse: Emerging Technology.* IEEE Computer Society Press, 1988.

[Ous96] John K. Ousterhout. *Graphical Applications with Tcl and Tk.* Professional Computing Series. M and T Books, 1996.

[Sch97] Fred B. Schneider. *On Concurrent Programming.* Graduate Texts in Computer Science. Springer-Verlag, 1997.

[SPL95] Alexandre Santoro, Woosang Park, and David C. Luckham. SPARC-V9 Architecture Specification with Rapide. Technical Report CSL–TR–95–677, Computer Systems Lab, Stanford University, September 1995.

[TGD97] Dimitrios Tombos, Andreas Geppert, and Klaus R. Dittrich. Semantics of reactive components in event-driven workflow execution. In *Proceedings of the Ninth International Conference on Advanced Information Systems Engineering*, pages 409–422, 1997.

[WY95] Steven Woods and Qiang Yang. Understanding as constraint satisfaction. In *Proceedings of the Seventh International Workshop on Computer Aided Software Design*, pages 318–327, 1995.