JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©**JOT, 2004**

Vol. 3, No. 10, 2004

SECG: The SCOOP-to-Eiffel Code Generator

Oleksandr Fuks, Jonathan S. Ostroff, Department of Computer Science, York University, Canada
Richard F. Paige, Department of Computer Science, University of York, York, U.K.

The Simple Concurrent Object-Oriented Programming (SCOOP) mechanism introduces inter-object concurrency to the Eiffel language, via addition of one new keyword, **separate**. We describe a general tool that takes a Eiffel program that uses the **separate** keyword and translates it into an Eiffel multi-threaded program that uses the Eiffel *THREAD* class. The resulting code is thereafter compatible with EiffelStudio and any other Eiffel compiler that provides the *THREAD* class.

1 INTRODUCTION AND BACKGROUND

Many mechanisms exist for introducing concurrency into object-oriented (OO) programming languages. These approaches support the use of multiple, perhaps distributed processors, each of which may be executing multiple processes. Different techniques are provided with the languages to support synchronisation, interruption, mutually exclusive access to object state, and atomic execution of routines.

Java [5] introduces concurrency via inheritance from special classes that introduce threads; additional compilers for Java have been developed for efficiently implementing Java's concurrency model, e.g., Jalapeno [1]. The Eiffel// project [2] provided a special class *PROCESS* that could be used to introduce new threads of execution. Jalloul [7] extends Eiffel by providing new language features for implementing critical regions and conditional critical regions; these new features are in turn implemented in a kernel sitting atop PVM. Variants of Smalltalk have been proposed [12, 9] for multi-threading. Similarly, C++ extensions such as Parallel-C++ [8] exist for parallel, distributed, and concurrent execution.

The Simple Concurrent Object-Oriented Programming (SCOOP) mechanism was proposed as a way to introduce inter-object concurrency into the Eiffel programming language [10]. The mechanism extends the Eiffel language by adding one keyword, **separate**, that can be applied to classes, entities, and formal routine arguments. Application of **separate** to a class indicates that the class is executing in its own thread of control; application of **separate** to entities or arguments indicate that these constructs are points of synchronisation, and can be shared among concurrent threads. This mechanism was implemented by Compton [4] by building upon the GNU SmartEiffel compiler and runtime system. No implementation for other versions of Eiffel, e.g., ISE EiffelStudio, exists, though much work is underway at ETH Zurich.

We describe a tool, called the SCOOP-to-Eiffel Code Generator (SECG), which translates Eiffel programs that use the SCOOP mechanism via **separate**, into Eiffel threaded applications that make use of the *THREAD* class which is packaged with many distributions of EiffelStudio. The results of applying the tool have been used successfully with EiffelStudio 5.2. SECG differs from Compton's implementation in that it does not rely on changes to a compiler (it translates SCOOP code into pure Eiffel) or a run-time system; thus, it can in theory be used with any version of Eiffel that provides an implementation of the *THREAD* class which conforms to ISE's specification.

The paper is organised as follows. We start with a brief overview of the SCOOP mechanism, as specified in [10], and summarise Eiffel's *THREAD* class, and then explain how the SECG translation tool works. We use two examples to illustrate the design and implementation of the tool. We then discuss limitations with SECG as implemented, and consider further work.

2 OVERVIEW OF SCOOP AND EIFFEL THREADS

SCOOP introduces concurrency to Eiffel by addition of the keyword **separate**; it is the responsibility of the underlying run-time system and compiler to deal with the subtle (and, in some cases, complicated) semantic problems introduced by the addition. The **separate** keyword may be attached to the definition of a class, or the declaration of an entity, or formal routine argument. Examples of the three types of attachments are as follows.

$$\textbf{separate class } ROOT \tag{1}$$

$$x : \textbf{separate } PROCESS \tag{2}$$

$$f(y : \textbf{separate } PROCESS) \tag{3}$$

A class that is declared as **separate** (as *ROOT* in (1)) cannot be declared as expanded or deferred; nor is its property of being **separate** inherited. A separate class executes in its own thread; thus, service requests (i.e., feature calls) to instances of a separate class may need to block or wait until the thread is available to execute the request.

An entity or argument declared as separate (e.g., as in (2) and (3) above) indicates that the data attached to the entity or argument may be shared between threads. Thus, synchronisation facilities must be provided so that, e.g., mutually exclusive writes to shared data take place. Entity x can only be declared as **separate** if *PROCESS* in (2) is not deferred or expanded.

SCOOP is based upon the notion of a *processor*, which defines a unit of execution in an OO system. When a separate object (defined in the sequel) is created, a new processor is also created to handle its processing. Thus, a processor is an autonomous thread of control capable of supporting *sequential* instruction execution [10]. A system in general may have many processors associated with it. Compton [4] introduces the notion of a *subsystem* – a model of a processor and the set of objects it operates on – to distinguish the execution of sequential and concurrent programs. In his terminology, a separate object is any object that is in a different subsystem.

Routine calls

In Eiffel, the standard syntax for routine calls is (i) $x.c(a)$ for a command c, which may change the state of the object attached to x, and (ii) $y := x.f(a)$ for a side-effect free function f. In sequential Eiffel, and in both cases, when executing the routine call, execution switches to the object attached to x, the routine executes, and (perhaps after storing a result), execution continues at the next instruction. Now suppose that either x is attached to a separate object, or that the type of x is separate. For the call $x.c(a)$, execution on the current object and x synchronise; x registers the fact that c was called and either starts execution of c immediately, or when the next opportunity arises. Then both the current call and $x.c(a)$ can proceed concurrently. If there are multiple pending requests for calls on x, they are queued and served in first-in-first-out order.

For case (ii), where a result is needed from a separate call, a restricted version of the *wait-by-necessity* mechanism of Caromel [3] is used, because the result of a call to $x.f(a)$ may not be available when the assignment $y := x.f(a)$ can take place. In SCOOP, further client calls on x will wait until the query call $x.f(a)$ has terminated.

Waiting

Eiffel introduces **require** and **ensure** clauses for specifying the pre- and postcondition of routines. In a sequential programming, a **require** clause specifies conditions that must be established and checked by the client of the routine; the **ensure** clause specifies conditions on the implementer of the routine. In a SCOOP Eiffel program, a **require** clause on a routine belonging to a separate object specifies a *wait* condition: if on a call to $x.r(a)$, where x is attached to a separate object, the routine's **require** clause is false, the processor associated with the object should wait until it is true before proceeding with routine execution.

Object reservation

There are many situations in a concurrent OO program where exclusive use of a separate object is required. In order to retain consistency and correctness, there must be some mechanism for stopping or pausing any interleaving of concurrent calls. SCOOP enables this by altering the semantics of argument passing. Consider the call

$$r(x : \textbf{separate}\ T1, \ldots, y : \textbf{separate}\ T2) \tag{4}$$

Exclusive locks should be obtained on x and y before the call to r starts; all locks must be obtained before the processor executes the call[1].

[1]In general, locks need only be obtained if a feature is called on an argument in the body of r.

Consistency rules

A SCOOP program may have both separate and non-separate objects. It is essential to guarantee that an entity declared as non-separate (e.g., $x : T$) can never be attached to a separate object; this could lead to race conditions and object inconsistency. In order to prevent this, Meyer introduces four consistency rules [10].

1. If the source of an attachment is separate, the destination entity must be separate as well.
2. If an actual argument of a separate call is of reference type, the corresponding formal argument must be separate.
3. If the source of an attachment is the result of a separate call to a function returning a separate type, the target must be separate.
4. If an actual argument or result of a separate call is of expanded type, its base class may not include any non-separate attribute of a reference type.

Eiffel Threads

A simple threading mechanism is provided as part of a library that comes with ISE's distributions of EiffelStudio. To make use of threads, i.e., to implement a class that defines an Eiffel thread, a developer writes a new class that inherits from the interface $THREAD$. This class provides the following fundamental routines:

- $execute$: the routine to be executed by the new thread. In general, this must be implemented by the developer.
- $join$: the calling thread waits for the current child thread to terminate.
- $launch$: initialise a new thread running $execute$.
- $join\_all$: the calling thread waits for all other threads to terminate.

The class $THREAD\_CONTROL$ provides control over thread execution. Typically the root class of an Eiffel application inherits from $THREAD\_CONTROL$, and uses its $join$ and $join\_all$ routines to manage execution of spawned child threads.

The library also provides basic concurrent functionality, particularly through the class $MUTEX$, which provides a synchronisation object.

In [4], the SCOOP proposal of Meyer is implemented in the framework of the GNU SmartEiffel compiler and run-time system. We now describe the SECG code generator, which translates SCOOP Eiffel programs that use the **separate** mechanism, into multi-threaded Eiffel applications.

3 THE SCOOP-TO-EIFFEL CODE GENERATOR

The SCOOP-to-Eiffel Code Generator (SECG) tool provides implicit support for the SCOOP proposal by translating an Eiffel program that makes use of **separate** classes, arguments, and entities, into one that makes use of threads and the Eiffel class *THREAD*, which is available with distributions of EiffelStudio. No changes to the EiffelStudio compiler or run-time system are needed, and all Eiffel programming constructs can be used, including **once** routines. In an informal sense, SECG implements a *refinement* of the SCOOP specification into Eiffel classes and statements that do not make use of **separate**; we discuss this further in the sequel.

The basic mechanism underlying SECG is to add mutexes and buffers to **separate** classes in order to keep track of pending requests made by clients to make use of services. Additional and similar changes are made to separate entities and separate arguments to introduce mutexes, allowing synchronisation and mutually exclusive access. Each separate class, when translated, inherits from *THREAD* and is provided with a buffer containing pending services requests (i.e., feature calls). The root class of the system simply executes all threads; each thread, indefinitely, removes a pending request for service and executes the request.

We first describe the general translation scheme used by SECG, and then illustrate its use with two examples.

The SECG tool accepts a single command-line parameter indicating the name of a *project* file. The project file specifies the names of all Eiffel classes (and thus, all `.e` files) to be included in the project. As well, the root class of the project must be specified with the keyword *root* prepended.

Using the information provided in the project file, the generator scans the files included in the system. The generator then produces the code as follows.

1. *THREAD_CONTROL* is added as a superclass of the root class. This provides the root of the application with control over thread execution. The root class is responsible for making sure that, when the application terminates, all pending service requests on all threads in the application have been handled.

2. All classes inherit from *EXCEPTIONS*.

3. *requests_pending* and *requests_pending_mutex* are added as attributes to the root class. The former attribute is used as a resource monitor for the root class, while the latter attribute synchronises access to the monitor (since clients may make service requests of the root class).

4. The following features are also declared and implemented in the root class.

   is_request_pending is used to determine if there are pending accesses to the root.

```
is_requests_pending : BOOLEAN is
do
  Result := true
  requests_pending_mutex.lock
  if requests_pending.is_equal(0) then
    Result := false
  end
  requests_pending_mutex.unlock
end
```

As well, a general-purpose *rescue* routine is provided to flag exceptional behaviour in the root class.

```
rescue_SCOOP(who_caused: STRING; what_caused: STRING) is
do
  io.put_string("Assertion violated in "+who_caused+": "+what_caused)
  raise("Assertion " + what_caused + " violated in " + who_caused)
end
```

5. Each class declared as **separate** inherits from *THREAD*; thus, each separate class has its own thread. The basic idea in translating a **separate** class is to provide a buffer for service requests (along with a mutex to ensure synchronised access).

   The following attributes are declared.

```
    requests_pending: INTEGER_REF
    requests_pending_mutex: MUTEX
    request_buffer: LINKED_LIST[TUPLE]
    request_buffer_mutex: MUTEX
    current_feature_args: TUPLE
    current_feature_name: STRING
```

   The attributes prefixed with *request* are used to ensure mutually exclusive access and also to buffer the requests for access; concurrent requests for service are, of course, queued. The attributes prefixed with *current* store the current feature (service) being requested and the arguments supplied to the call. Requests for services are stored as tuples, containing the target of the service request and the name of the service requested, encoded as a string. Decoding takes place in the *execute* routine of the thread.

   Additional routines must be added to each separate class in order to provide mutually exclusive access and FIFO buffering of service requests. *is_requests_pending* and *rescue_SCOOP* are identical to the ones defined in the root class above; we do not repeat their definitions here. The routine *set_feature_to_do* simulates a feature call that is pending. It will first obtain the lock on the pending requests mutex, and increase the number of pending requests. The buffer of pending requests is extended with suitable arguments.

```
set_feature_to_do(feature_params_arg: TUPLE) is
do
  requests_pending_mutex.lock
  requests_pending.copy(requests_pending + 1)
  requests_pending_mutex.unlock
  request_buffer_mutex.lock
  request_buffer.extend(feature_params_arg)
  request_buffer_mutex.unlock
end
```

get_feature_to_do removes a pending request from the buffer, if one exists; otherwise, a dummy empty request is returned, which can be used as a termination signal to a controlling thread, e.g., the root.

```
get_feature_to_do: TUPLE is
do
  request_buffer_mutex.lock
  if not request_buffer.is_empty then
    Result := request_buffer.first
  else
    Result := [Current, "NOTHING"]
  end
  request_buffer_mutex.unlock
end
```

Separate entities and arguments

Changes must also be made to entities declared as separate. The declaration

```
x: separate SOME_TYPE
```

in a SCOOP Eiffel program is replaced by SECG with the declarations

```
x: SOME_TYPE
x_mutex: MUTEX
```

A similar addition is made for separate arguments: a mutex is added for each separate argument, and the **separate** keyword is removed. This is illustrated in the examples in the next section.

Creation procedures

Given that separate classes and entities are being replaced with threads, buffers, and mutexes, the creation procedures of translated separate classes must be extended to initialise mutexes and service request buffers accordingly. In the declaration of the creation procedures of separate classes, two arguments are added:

```
requests_pending_arg: INTEGER_REF
requests_pending_mutex_arg: MUTEX
```

Initialisation is also provided for these attributes in all creation procedures of separate classes. At the start of the creation procedure of the root class the following instructions are added:

```
create requests_pending_mutex.default_create
requests_pending := 1
```

At the end of this creation procedure we add instructions which guarantee correct completion of the application. All requests for service that are still pending are removed from the buffer, and then the routine *join_all* of class *THREAD* is called; the root class will then wait (and termination of the application will therefore wait) until all threads have finished execution.

```
from
  requests_pending_mutex.lock
  requests_pending.copy(requests_pending -1)
  requests_pending_mutex.unlock
until not is_requests_pending
loop end
join_all
```

In the creation procedures of separate classes we add the following instructions, which initialise the pending services request buffer to empty, and initialise the mutex for the class.

```
requests_pending := requests_pending_arg
requests_pending_mutex:= requests_pending_mutex_arg
current_feature_name := "NOTHING"
create current_feature_args.make
create request_buffer.make
create request_buffer_mutex.default_create
```

Calls

Finally, we can translate calls to routines. We substitute calls to features of formerly separate classes as follows. The call

```
  p.some_feature(d)
```

where *p* is an entity of a separate class, and *d* is a separate reference, is translated to the call

```
p.set_feature_to_do([Current, "SOME_FEATURE_STRING", d, d_mutex])
```

The first argument indicates the target of the feature call; the second is a string encoding of the feature being requested. Note that a mutex is supplied with the separate argument *d* so that mutually exclusive access can be arranged.

One question remains: the above translation effectively *buffers* service requests. So when do service requests actually get processed, and features called? This is carried out in the routine *execute*, which must be implemented by the translation of every **separate** class; *execute* is a deferred routine inherited from *THREAD*. Effectively, all that *execute* does is remove a tuple from the request buffer, decodes the feature to be executed, and executes it. We illustrate this in the examples.

Finally, SECG automatically places lock/unlock instructions where necessary, i.e., when attempting to write to formerly separate entities. This is illustrated in more detail in the next sections, where examples show how the conversion process works.

4 ONE-ZERO EXAMPLE

Our first example is called one-zero; it is intentionally simple in order to illustrate the basic conversion process. We assume that we have two classes, *PROCESS* and *DATA*. *PROCESS* is a separate class, while *DATA* is used to represent shared data; thus, access to an entity of type *DATA* should be synchronised in some way. We will create three entities of class *PROCESS*, which will access a synchronised entity of type *DATA*. We will use the class *PROCESS* further in the next section, where we show the effect of applying SECG to it.

SCOOP source

Consider the following SCOOP Eiffel program, consisting of a single root class. The program creates tree entities of separate class *PROCESS*, which will access the separate entity of type *DATA*. The details of class *PROCESS* are in the next section, but for now it suffices to know that all the *run* routine of *PROCESS* can do is either set the value stored in d to 0 or 1, or print the stored value in d.

```
class ROOT_CLASS
creation make

feature
  d: separate DATA
  p1, p2, p3: PROCESS -- separate class

make is -- start three processes
do
  io.putstring ("Test threads%N")
  create d.make
  create p1.make(d,0,"First")
  create p2.make(d,1,"Second")
  create p3.make(d,2,"Third")
  p1.run
  p2.run
  p3.run
end
end -- class ROOT_CLASS
```

Generated source

After applying SECG to the above class, the following result is generated. First, inheritance from *THREAD_CONTROL* and *EXCEPTIONS* is added. Further, a mutex is added for separate entity d. Since we have several threads (because each process $p1, p2$, and $p3$ are separate entities) – each of which can place service requests to the others – we need to know when requests were executed and if there is a need to continue thread execution. We thus introduce variables to keep track of pending requests (and their number). Once all requests have been executed (i.e., *requests_pending* is zero), thread execution can terminate. The following source is therefore generated.

```
class ROOT_CLASS
inherit
  EXCEPTIONS
  THREAD_CONTROL

creation make

feature
  d_mutex: MUTEX
  requests_pending: INTEGER_REF
  requests_pending_mutex: MUTEX

  is_requests_pending:BOOLEAN is
  do
    Result := true
    requests_pending_mutex.lock
    if requests_pending.is_equal(0) then
      Result := false
    end
    requests_pending_mutex.unlock
  end

  rescue_SCOOP(who_caused:STRING;what_caused:STRING) is
  do
    io.put_string("Assertion violated in " + who_caused + ": " + what_caused)
    raise("Assertion " + what_caused + " violated in " + who_caused)
  end
```

The attributes from the source file are translated directly, with the **separate** keyword removed.

```
d:  DATA
p1, p2, p3: PROCESS
```

make must be modified according to the translation scheme described in the previous section. Its purpose is to start the three processes. First, it initialises *requests_pending* to

1 since the creation procedure is a service that can make further requests. It then initialises the attribute *d* and its mutex.

```
make is
do
  create requests_pending_mutex.default_create
  requests_pending := 1

  io.putstring ("Test threads%N");
  create d_mutex.default_create
  d_mutex.lock
  create d.make
  d_mutex.unlock
```

Next, we translate the statements contained in the body of the original *make* procedure. The statements are **create** statements and process *run* statements. For translating the **create** statements, we add *requests_pending* and *requests_pending_mutex* parameters, and also *d_mutex* since the attribute *d* is declared as separate and we may need to synchronise access to it. After creating each *PROCESS* object, we launch the corresponding thread.

```
create p1.make(d, d_mutex, 0,"First", requests_pending, requests_pending_mutex)
p1.launch
create p2.make(d, d_mutex, 1,"Second", requests_pending, requests_pending_mutex)
p2.launch
create p3.make(d, d_mutex, 2,"Third", requests_pending, requests_pending_mutex)
p3.launch
```

We must next translate the *run* feature calls. As with any feature call, it is translated to invocations of thread *set_feature_to_do* calls, which effectively inform the thread that a service request of the feature specified as a parameter is being made; the thread can then buffer the service request and carry it out as soon as possible.

```
  p1.set_feature_to_do([Current,"RUN_STRING"])
  p2.set_feature_to_do([Current,"RUN_STRING"])
  p3.set_feature_to_do([Current,"RUN_STRING"])
```

Finally, all pending requests must be removed from the buffer for the class, and the root class thread must wait until all other threads have terminated, before it can terminate

```
from
  requests_pending_mutex.lock
  requests_pending.copy(requests_pending - 1)
  requests_pending_mutex.unlock
until not is_requests_pending
loop end
join_all
end -- make
end -- class ROOT_CLASS
```

The above translated program compiles and executes under EiffelStudio 5.2.

5 EXAMPLE: CLASS *PROCESS*

The example of the preceding section makes use of the separate class *PROCESS*. We now show how SECG translates this separate class into a threaded Eiffel class. *PROCESS* is a straightforward class, possessing a name, an option, and shared data. When the process runs, it can do one of three things: sets its shared data to 0; to 1; or view and print its data. Here is its source.

```
separate class PROCESS
creation make
feature
  option: INTEGER
  data: separate DATA
  name: STRING

  make(d: separate DATA; opt:INTEGER; n:STRING) is
  do
    data := d
    option := opt
    name := n
  end

  run is
  local i:INTEGER
  do
  from until false
  loop
    if option = 0 then
      data.zero -- set data to zero
    elseif option = 1 then
      data.one -- set data to one
    else data.view; print_me
    end
  end
  end



  print_me is
  do
    print("%N" + name + " just ran" + "%N")
  end
end -- class PROCESS
```

SECG must carry out several tasks in translating this class: it must implement mutexes for separate entities, add inheritance clauses for the separate class, and translate the separate arguments in *make*. An implementation must also be provided for the *execute* feature, which must be implemented in any class that inherits from *THREAD*. *execute* simply takes requests from the pending buffer and executes the corresponding feature (either *run* or *print_me*). Here is a snapshot of the translation.

```
class PROCESS
inherit
  THREAD
  EXCEPTIONS
creation make
feature
  execute is
  do
    from
    until not is_requests_pending
    loop
      current_feature_args := get_feature_to_do
      current_feature_name ?= current_feature_args.item(2)
      if not current_feature_name.is_equal("NOTHING") then
        if current_feature_name.is_equal("RUN_STRING") then
          run
        end
        if current_feature_name.is_equal("PRINT_ME_STRING") then
          print_me
        end
        requests_pending_mutex.lock
        requests_pending.copy(requests_pending - 1)
        requests_pending_mutex.unlock
        request_buffer_mutex.lock
        request_buffer.start
        request_buffer.remove
        request_buffer_mutex.unlock
      end
    end
  end
```

As discussed in Section 3, a number of features will be automatically added by SECG for keeping track of pending requests to a (translated) separate object, to keep track of which feature is being called by a thread, and to handle exceptions. These features, such as *requests_pending* and *set_feature_to_do*, are added to the translate of *PROCESS* at this state, as described in Section 3.

Next, SECG copies over attributes from the separate class *PROCESS* into the threaded version; this includes *option*, *data*, and *name*. The creation procedure *make* is then translated, adding three new arguments: *d_mutex* (to handle mutually exclusive access to the data), *requests_pending*, and a mutex. Finally, *run* can be translated, and at this point we can illustrate the addition of locking and unlocking of mutexes, which must be before and after accessing any shared (separate) entities.

```
run is
local i:INTEGER
do
  from
  until false
  loop
    if option = 0 then
      data_mutex.lock
      data.zero
      data_mutex.unlock
    elseif option = 1 then
      data_mutex.lock
      data.one
      data_mutex.unlock
    else
      data_mutex.lock
      data.view
      data_mutex.unlock
      print_me
    end
  end
end
```

Limitations

A SCOOP program that is translated using SECG is not guaranteed to be deadlock free: if a programmer misuses shared data or synchronised processes, it is not difficult to introduce deadlock (or livelock) among threads. It is not clear, based on [10], to see how deadlock freedom can be guaranteed for SCOOP programs.

The SCOOP proposal in [10] allows *local* variables to be declared as **separate**. This is not permitted in SECG; any locals declared as separate will not be translated correctly, nor will the resulting program compile. An entity declared as **local** has its lifecycle linked to that of the execution of its enclosing routine. Once the routine terminates, any object attached to the entity will be destroyed. An entity declared as separate is intended to be (potentially) shared by multiple threads; thus, it seems that declarations of **local** and **separate** are incompatible. It remains for future work to investigate whether the two mechanisms can be reconciled.

There are no further limitations with SECG: any valid Eiffel constructs, including **once** routines and **expanded** types can be used. Because SECG is a pre-processor, and because it implements **separate** classes and entities in terms of *THREAD*s, instead of modifying the underlying run-time system, it should not be affected by changes to the Eiffel language, e.g., additions of new constructs.

Soundness

The soundness of SECG has not been proven, though the tool has been tested extensively on a number of case studies. Soundness could be proven by appealing to the Eiffel Refinement Calculus (ERC) [11]. This calculus provides a formal semantics for a subset of Eiffel (including feature calls and reference types). The calculus currently supports real-time specification, but it could be extended to concurrency and multi-threading; the calculus is built atop Hehner's predicative programming calculus [6], which supports concurrency and communicating processes. The calculus could then be used to give a formal semantics to **separate** classes and entities. Thereafter, it could be shown that a class produced by SECG *refines* a separate class in SCOOP.

6 CONCLUSIONS

We have given an overview of the SECG tool, which implements the SCOOP concurrency proposal for Eiffel by translating Eiffel programs that use **separate** entities and classes into threaded applications. Two examples have demonstrated the process, and limitations with the tool have been discussed. With some work and tuning for efficiency, a mechanism like SECG could form the basis for an industrial-quality implementation of the SCOOP mechanism in open-source Eiffel compilers.

The latest alpha version of SECG can be obtained from the authors. SECG is itself written in Eiffel, and has been tested and evaluated under ISE EiffelStudio 5.2.

REFERENCES

[1] B. Alpern et al. The Jalapeno Virtual Machine. *IBM Systems Journal* 39(1), 2000.

[2] I. Attali and D. Caromel. Formal Properties of the Eiffel// Model. *Parallel and Distributed Objects*, 1999.

[3] D. Caromel. Towards a method of object-oriented concurrent programming. *Comm. ACM* 36(9), September 1993.

[4] M. Compton. *SCOOP: an Investigation of Concurrency in Eiffel*, MSc Thesis, Australian National University, 2000.

[5] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*, Second Edition, AWL, 2000.

[6] E.C.R. Hehner. *A Practical Theory of Programming*, Second Edition, Springer-Verlag, 2003.

[7] G. Jalloul. Communicating Sequential Systems. *Journal of Object-Oriented Programming*, 2000.

[8] C.-H. Jo, C.-J. Lee, and J. Son. A realization of a concurrent object-oriented programming language. In *Proc. ACM Symposium on Applied Computing* 1998, ACM Press, 1998.

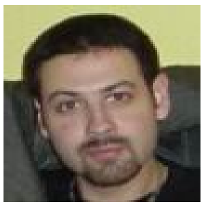
[9] D. Konstantas, O. Nierstrasz, and M. Papthomas. An implementation of Hybrid, a concurrent object-oriented language. Technical Report, University of Geneva, June 1998.

[10] B. Meyer. *Object-Oriented Software Construction*, Second Edition, Prentice-Hall, 1997.

[11] R. Paige and J. Ostroff. ERC: an Object-Oriented Refinement Calculus for Eiffel, to appear in *Formal Aspects of Computing*, Springer-Verlag, 2004. Draft available at http://www.cs.yorku.ca/techreports/2001.

[12] Y. Yokote and M. Tokoro. Experience and evolution of Concurrent Smalltalk. *SIGPLAN Notices* 22, October 1987.

ABOUT THE AUTHORS



Oleksandr Fuks is a Masters student at York University, Toronto, Canada, expecting to complete his thesis in 2003. Email: nati@cs.yorku.ca.

Jonathan Ostroff is an associate professor at York University, Toronto, Canada, where he leads research on object-oriented design, formal methods, and real-time software development. Email: jonathan@cs.yorku.ca.

Richard Paige is a lecturer at the University of York, UK, where he works with the High-Integrity Systems Group and is a co-leader of the Software and Systems Modelling Team. He carries out research in object-oriented development, formal methods, security, and distributed systems. He completed his PhD in Computer Science at the University of Toronto in 1997. Email: paige@cs.york.ac.uk.