Developing multi-threaded Java applications from high-level models in the PEPA stochastic process algebra

Stephen Gilmore* and Kris Powell

Laboratory for Foundations of Computer Science, The University of Edinburgh, Edinburgh EH9 3JZ, Scotland. Email stg@lfcs.ed.ac.uk, kris@krispowell.net

Abstract. The design and implementation of concurrent applications is more challenging than that of sequential applications. The aim of this work is to address this challenge by producing an application which can generate skeleton Java systems from a high-level PEPA modelling language description. By automating the process of translating the design into a Java skeleton system, the result will maintain the performance and behavioural characteristics of the model, providing a sound framework for completing the concurrent application. This method accelerates the process of initial implementation whilst ensuring the system characterisics remain true to the high-level model.

1 Introduction

The design and implementation of distributed and concurrent systems has proved problematical, as the interaction of multiple executing processes can lead to unexpected and unwanted behaviour, such as deadlock, live-lock and starvation. The objective of this work is to develop a methodology and supporting tools that automate part of the process in order to remove some of the pitfalls associated with the design of higher performance concurrent and distributed systems.

One common technique is to use modelling languages to aid the design process as they allow a particular design to be tested and refined in order to deliver maximum performance whilst eliminating problems such as hotspots, bottlenecks and deadlock. However, because there is usually no clear parallel between modelling language constructs and the facilities provided by programming languages, there is the danger that the design of the system may veer away from the design suggested by the model once actual implementation of the system begins.

Automating the process of the model's translation to a 'skeleton' implementation means the desirable performance and behavioural properties of the model are maintained, providing a sound framework for completing the implementation of the system whilst ensuring the system characterisics remain true to the high-level model.

Using Performance Evaluation Process Algebra (PEPA) [4] and Java, the aim of this work is to produce three things:

* Corresponding author. Web page: http://homepages.inf.ed.ac.uk/stg/ Tel: +44 131 650 5189. Fax: +44 131 667 7209.

1. A PEPA2Java API that defines an interface specifying PEPA constructs and their behaviour in Java terms (e.g. a *Component* Java class with methods such as *choice* to mimic the behaviour of PEPA *components* and the choice behaviour of these components.)
   One of the reasons for defining the API in such a way is that it would allow multiple implementations providing, for example, distributed in place of concurrent execution of a model.
2. A package that implements the PEPA2Java API to run the PEPA system as a Java application executing concurrent *threads* simulating the behaviour of the components of the model.
   The models produced will implement certain abstract classes and methods defined in the API, and use the defined methods inherited to handle the actual "mechanics" of running the simulation.
3. A *Translator* utility which automates the translation of a PEPA model into objects and methods of the type defined by the PEPA2Java API.
   The result is a set of Java classes that, when compiled and executed, will utilize a PEPA2Java API implementation to execute the model as a Java application.

PEPA can be used to evaluate performance as well as testing for correct functional behaviour in a model. It also models parallel composition and synchronisation behaviour. These attributes make it an ideal choice for this work. Briefly, PEPA is a timed process algebra where the prefixes are pairs which specify the *type* and *rate* of an activity in the form $(\alpha, r).P$, where P is the continuation of the execution. Choices between different possible behaviours are denoted by $P + Q$ and resolved operationally by a rate condition. Synchronisations between P and Q over the activity types in the set L are denoted by $P \underset{L}{\bowtie} Q$. $P \parallel Q$ is the degenerate case where L is empty, and so there is no synchronisation between P and Q at all. P/L is like P except that the activites in the set L are hidden, becoming silent τ actions which cannot be synchronised upon. For more on PEPA see [4].

Java is the target language because of its flexibility, its widespread adoption, its portability across platforms and its rich set of standard libraries. Additionally, it has native support for multiple threading and remote method invocation. As an object oriented language, it can take advantage of inheritance, making it simpler to run a PEPA model as a Java process whilst hiding all the underlying mechanics from the user, as well as allowing different implementations of the interface to run the same PEPA2Java skeletons. Also, this should make it easier to flesh-out and extend the generated skeletons by limiting the generated "clutter".

The methodology of separating the model from the PEPA-simulating code by using inheritance and also a tree-model used to represent synchronization behaviour distinguishes this work from previous work on automatic program generation from PEPA models (for example, using Ada [3]). Notably, thread-based programming in Java is very different from concurrent programming with Ada's tasks.

2 Compiling PEPA to Java

The goal of this work is to be able to run any PEPA model as a Java application. The procedure of finding a mapping from PEPA to Java often involved changes to the design of our translator as limitations were discovered. Many improvements to the API were made during the implementation stage, forcing a re-evaluation of how best to compile PEPA into Java. Documented here is the final set of algorithms and concepts that present a method of executing a PEPA model as a running Java system, compensating for most of the limitations which this entails.

2.1 Methodology

We employ a direct mapping between the "objects" of PEPA (e.g. *components* and *actions*) and objects in Java. Using inheritance, the idea is to provide a public interface that defines a number of classes and methods that specify PEPA-equivalents for Java. Models are run by extending these classes, defining their abstract methods, and creating instances of the appropriate objects which utilize the superclasses' framework and defined methods. One of the benefits of the API is that it is general and flexible enough to work with different implementations, each providing different functionality. The API-implementing packages can be used interchangeably to provide different functionality for executing a PEPA model.

One important restriction to note is that PEPA's *Hiding* combinator is not present in this API. Whilst it was realised that the hiding construct is of great value during abstract model design and evaluation, its importance in an actual system implementation is not as clear. In particular, it does not correspond to the traditional programming language encapsulation provided by Java's package mechanism and public and private modifiers. Encapsulation in Java is a compile-time restriction on visibility of names at the class level whereas hiding in PEPA is applied at the instance level to instantiations of processes. Hiding is used to simplify complex models, but the process of top-down implementation works in the opposite direction—adding the underlying complexity abstracted away in modelling languages. Accordingly, there is no equivalent for PEPA's hiding combinator in the API and the translator will not translate any model which uses hiding. It would be possible to implement hiding by a combination of duplicating component definitions and renaming but such an approach would increase the complexity of the generated Java and would not promote code reuse.

2.2 Branching and Race Conditions

One critical decision when translating from a PEPA model to an actual implementation relates to branching. In models, the choice of which branch to execute is simplified. In the case of PEPA, it is determined by a race condition of randomly sampled, exponentially distributed rates. However, in an implemented system, branching is decided by the system's rules and history—namely, it is decided algorithmically. The generated skeleton's behaviour is probabilistic and rates need to be carefully chosen in order for the results of performance analysis on the model to be applicable to the implementation. Nevertheless, the skeleton is only the starting point: the main task of implementation is to replace probabilistic with deterministic branching.

There are two forms of branching in PEPA, both determined by race conditions. Choice constructs are the obvious form. The more subtle form is the case where multiple components are all competing for the chance to cooperate on some shared action where not all may cooperate at the same time. An example of this is two clients trying to synchronise with one server on a critical section.

The PEPA race condition is a form of speculative execution of multiple branches: all branches are executed and the first one to finish is the winner. The winning branch then dictates the ensuing behaviour of the component, whilst all other executions are aborted. Speculative execution is *not* an option when producing skeletons that are to become actual applications: it would mean the forking of execution and aborting of actions mid-task, which would make the skeleton impossible to "flesh out" and lead to inconsistency.

Simulating the race condition as follows allows some certainty:

– once committed to an action, a component will not do anything else until the action has completed;
– once started, an action must complete;
– if a branch has been chosen, the other branches will definitely not be chosen, and will therefore not be able to affect the system.

These refinements allow us to make progress from an initial high-level model towards a finished implementation.

3 Analysing PEPA terms for communication behaviour

Data structures called *and-or synchronization trees* are created to address the challenge of representing PEPA synchronization types in Java. Each action holds the root of such a tree, which is made up of synchronization set node objects. The synchronisation set node class has the activity class as an (abstract) subclass with individual and shared activities as (concrete) subclasses of that.

Synchronization set trees connect actions to their participating activities. There are three types of synchronization set node objects: two of these (`AND` nodes and `OR` nodes) are used as internal nodes in the tree and the third (activity objects) is used as the leaves. Each branch node has a left child and a right child. An example will help to illustrate the concept. Consider the PEPA composition:

$$(\mathit{Client} \parallel \mathit{Client}) \underset{\{\mathit{serve}\}}{\bowtie} \mathit{Server}$$

From this, we can see that there are two cooperations: both include the *Server* component, and either one or the other of the two *Client* components. This would yield the synchronization tree shown in Figure 1. The most important function of the synchronization tree is to determine whether the action may run or not. An action is constantly waiting to run—each time an activity *joins* it, using Java's `Thread.join` method, the action is notified (through the Waiter class) and will re-check the status of its synchronization tree to see whether it may run. Checking is done by calling `isLocked` on the root node of the tree. The method is recursively called on all nodes. `AND` nodes return true if both the left and the right child return true, whereas `OR` nodes return true if either (or both) of the children return true. The tree construct is equivalent to the PEPA composition because the `serve` action requires the `Server` component and either of the two `Client` components to cooperate before it can execute.

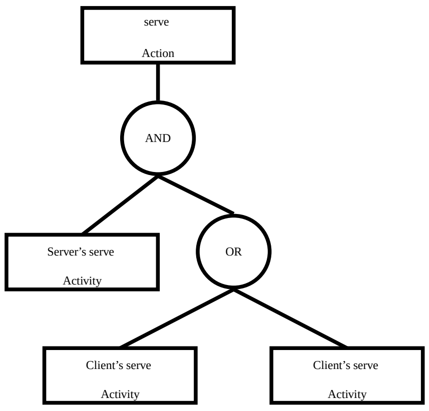

Fig. 1. A simple synchronization tree

A shared activity will most likely need to wait for additional components to join before the action can run. Even then, if a component does join an action, it may or may not be chosen as a participant if there are multiple possible combinations of cooperators for an action: in this case, the `Server` component will always participate whenever the `serve` action is run. However, only one of the two `Client` components may join each run. Therefore, if both `Client` components are waiting for `serve`, one must be chosen to be a *Runner* whilst the other will need to wait for a future run of the action: the `setRunners(actionThread)` method traverses the tree and picks the "fastest" subset which is ready to go. The chosen Activities will call `actionThread.join` to lock their threads to the action's Thread object. Also, the `setRunners` method *returns* the determining rate of the subset it has chosen, which will specify the length of time that the action must pause for "executing." Calling `setRunners` on an `AND` node will recursively call the method on both the node's children, whereas calling it on an `OR` node will cause it to be recursively called on only one of the node's children. Which `OR` child is chosen is determined firstly by whether one or both of the children return true for `isLocked`. If only one of the two subtrees returns true for `isLocked`, then `setRunners` is called on that subtree. If both are locked, then the faster branch will be returned. If both branches return the same speed (i.e. because they both run at rate $\top$), then the choice is decided by a weighted random function: the chance of `setRunners` being called on the left branch is equal to the proportion of leaves in the left subtree over the total number of leaves in the two subtrees. To avoid being continually passed over, the rates of the members of the sets which are ready but are *not* chosen as participants have their rates resampled each time the action runs.

The Translator must initially analyze the PEPA composition expression to discover the synchonization sets, and generate the appropriate PEPA2Java commands. The PEPA composition expression returned by the parser is of a tree form. For this part, consider a slightly different model that better demonstrates the algorithm used:

$$A \underset{\{m\}}{\bowtie} B \underset{\{n\}}{\bowtie} C$$

The parser will return the tree as shown in Figure 2. For PEPA2Java, a `setSynch`

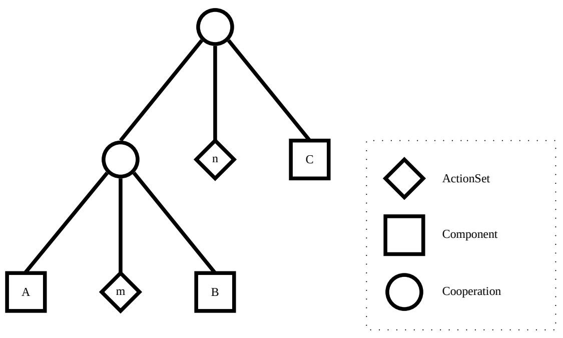

Fig. 2. The data structure created from $A \underset{\{m\}}{\bowtie} B \underset{\{n\}}{\bowtie} C$

method call must be made by each action, to create its synchronisation tree. To generate this method call, the translating algorithm goes through two stages— create a full synchronisation set tree, and then prune the tree. The four steps for generating the full synchronisation tree from the composition expression tree are (for each action):

1. Set a SharedAction flag to false.
2. Start at the top of the composition expression tree (fig. 2).
3. If the current node is a *Cooperation* node:
- If the current action is in this node's ActionSet, return a new AndNode to our synchronisation tree. Also, set the SharedAction flag to true.
- If the current action is not in this node's ActionSet, return a new OrNode to our synchronization tree.
- Set the left and right child to be the return result of a recursive descent into those nodes.
4. If the current node is a component identifier, return the reference to it.

For action m, this would return the unpruned tree shown in Figure 3. The next step is to prune the tree to contain only components that cooperate on this action. If the SharedAction flag is still set to false, this is an individual action which requires no cooperation— the synchronization tree is scrapped. Otherwise, we set the action's synchronization tree to be the result of the method `unpruned_root.prune`, which will return a pruned version of the tree. The three steps in the pruning algorithm are:

1. Call the `contains(thisAction)` method on the left child node. This method returns true if this sub-tree contains a component that contains an activity that participates in this action.
- If the method returns false, return the result of `right_node.prune`— this node, and its left sub-tree have been pruned. The ends the algorithm for this node.
2. Call the `contains(thisAction)` method on the right child node.

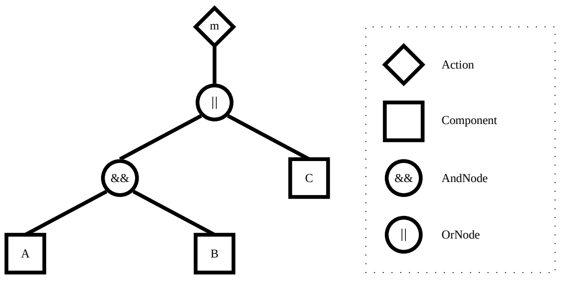

Fig. 3. Action m's unpruned synchronization tree for $A \underset{\{m\}}{\bowtie} B \underset{\{n\}}{\bowtie} C$

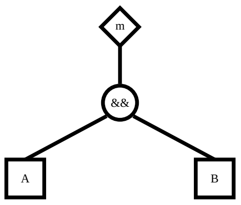

Fig. 4. Action m's pruned synchronization tree for $A \underset{\{m\}}{\bowtie} B \underset{\{n\}}{\bowtie} C$

- If the method returns false, return the result of `left_node.prune`.
- If no result has been returned, then both children must have returned true for `contains`. The left node is set to the result of `left.prune` and set the right node to `right.prune`.

3. Finally, return `this` as the result of `this.prune`— this node has not been pruned.

Action *m*'s resulting pruned tree is shown in Figure 4. Because the right child of the OrNode did not contain any activities participating in *m*, but its left subtree did, the left subtree was returned as the result of calling `prune` on the root. The OrNode and its right subtree were dropped. The left subtree survived intact because both its children contained components that participate in *m*. Performing these two algorithms on all the actions in the model will create the synchronization set trees needed to define the PEPA2Java API's abstract `initSynchs` method in the PepaSystem object.

4 An Implementation of the PEPA2Java API

The Simulator and Barebones packages are the concurrent system implementations of the PEPA2Java API. Their functionality (and their source code) is for the most part identical, except that the Simulator package includes extensions that give the ability to display the state of the system. This is useful for debugging purposes and tracking the execution of a model.

Central to the implementation is the PepaSystem class– it specifies three abstract methods which must be defined by implementing models. These are `createComponents`, `createActions` and `initSynchs`. The PepaSystem object creates the *Components* and their *Scripts*, the *Actions* and their *synchronisation sets*, initialises all these objects and starts the whole system running. In the case of the Simulator package, it will also control the GUI and the debugging messages.

The most important part of the implementation deals with ensuring that the Java skeleton of a model runs as the PEPA model dictates it should. However, there is no native support for the concept of synchronising multiple threads of execution for cooperation on an action. Therefore, much of the challenge lies in getting Java to behave in this way, as well as deciding the branching. The Action class, specified later in this section, ensures the proper synchronisation of multiple threads on cooperating sections. To accomplish this, the Action class, and also the Component class, make use of two simple concurrency control helper classes: the Waiter class and the Lock class (specified later).

The Waiter class provides *synchronised* wrapper methods around the `wait` and `notifyAll` methods inherited from `java.lang.Object`. It enables objects to wait for conditions becoming true, only checking conditions when another object notifies it that there has been some relevant change. (For example, when a component is waiting on any branch to become available for running in a choice block, or when an action is waiting for cooperating components to join.) This also ensures any user-defined subclass cannot inadvertently lock or release the object monitor and interfere with the functioning of the implementation.

A *Rate* class provides methods for accessing an exponentially distributed random number generator, which is used to calculate the period that an action pauses for, as well as determining race conditions in branching. Additionally, they can represent unspecified rates for passive participation.

The race condition is simulated in the following manner in the Simulator and Barebones packages:

1. Rate samples are decided *a priori*—the branch with the smallest sleep-time will be chosen. This is the case for both types of branching—choice blocks and multiple components competing to join an action.
2. Losing branches are resampled to simulate speculative execution—this prevents branches that get a high sleep-time from being passed over repeatedly because of one “unlucky” sample.
3. The winning branch will pause for its sleep-time, resample and then execute its next behaviour.
4. Before a branch can be chosen, it must be ready to execute—all the members of its synchronisation set must be prepared to participate. If no branches are ready, the system will execute the first branch that becomes so. Choosing commits the system to execute that branch—it cannot be interrupted by a faster branch that becomes available later.
5. Because the first ready branch wins, with no chance of faster branches “overtaking”, the choice method necessarily needs a pause (set as a constant in this implementa-

tion) to allow participants of shared actions to join. Otherwise, individual actions would unfairly dominate choice branching as they are *always* ready.

Each component object represents one of the components as defined in the composition expression of the PEPA model, and is run as a separate Java thread. A component class on its own defines no behaviour—for this, the CompScript interface is used. Every component has a reference to one or more CompScript objects. The `run` method in the Component class calls the current CompScript's `actions` method. This executes the behaviour of the component for one particular definition (for example, a series of prefix or choice combinators). The `CompScript.actions` method ends by returning a reference to a CompScript object (possibly itself). The `Component.run` method will then in turn execute the `actions` method of the returned CompScript. This mechanism enables components to take on many varying behaviours as defined by scripts, whilst their synchronisation sets and representation remain constant—hence, a Component instance is equivalent to a PEPA component as defined in the composition expression and a Component's CompScripts are equivalent to sequential process definitions in a PEPA model.

This execution mechanism of Components and CompScripts deploys a structured continuation-passing style to simulate PEPA recursive processes on an unmodified JVM without overflowing the run-time stack or generating any terminated Java threads which need to be reclaimed by the system's built-in garbage collector. Similar techniques (known as "trampolining" algorithms) are used to compile functional programming languages for the JVM.

The *Action* class defines the PEPA actions of the model, in which *Components* may take part (as PEPA activities). Each shared action of the model is represented by an action object, which implements the `Runnable` interface. It also holds the root to a synchronisation tree which is used to determine whether the action is ready to run or not. An action steps through five stages:

1. **Wait for** `noRunners`: The action object waits until all component members of its synchronisation set have their `runner` flag set to false—this indicates that the components know that the previous run of the action has completed, which is a prerequisite for the next run. This is necessary because if an action is very short there is the possibility in a multi-threaded execution environment that one or more of the last chosen component threads may not have resumed execution since the last run. One of those components might still be waiting for the already executed action to run whereas other components which are aware that the last action already ran might now be waiting for the next action to occur. Therefore, the action must pause between runs until all components have acknowledged its execution.
2. **Wait for** `allLocked`: Next, the action will pause until the `allLocked` method, called on the root of the syncronization set tree, returns true—this happens when all the component members of the synchronisation set have joined and committed to the running of the action, indicating that the necessary cooperation is available.
3. **Set the Runners**: Now the action is ready to be run, it notifies the members of a particular cooperation set that they have been chosen to participate in this run of the action. These set their `runner` flags to true, and join the action thread, meaning they are committed to cooperating.

4. **Start the Action**: The action calls its `action` method, which will cause the thread to sleep for a number of milliseconds set by the action's determining rate, calculated from the rates of all the participating activities. In the final implementation, the sleep action would be replaced by the actual functionality required.
5. **Reset**: The `reset` method is the final command of this run of the action—its reference (`this`) is passed to a new thread object as a *Runnable* argument. The action's rate is then reset to unspecified, and the action will `start` the newly created thread. With this, the old thread previously running the action will die, releasing the joined components, which will immediately set their `runner` flags back to false, signalling they are aware that the current run of this action has completed.

The `Activity` class provides methods for component objects to interface with actions. An activity is joined and commmitted to when a component calls its `join` or `joinAt` method. Conceptually, there is a subtle difference between activities as specified in PEPA and PEPA2Java. Namely, every component participating in an action holds one PEPA2Java activity per action—if a component joins activities participating in the same action at different rates, they will be represented by a single activity object in the component. This is known as an *internal choice*, and represents the cases where either the same action may be performed at different rates at different occasions or there is a silent activity which has a rate but does not have a name [2]. The component will simply *join* the activity at different rates at different times. This is done by changing the rate object reference which the activity holds. The `joinAt` method is identical to the `join` method except that it allows the caller to specify the rate at which the activity should be performed. It is preferred because it closely mirrors PEPA's *Prefix* semantics.

There are two sub-classes of the Activity class, SharedActiv and IndivActiv. SharedActiv is used to execute activities which require synchronization between multiple components.

The IndivActiv class is used for executing individual activities—those without synchronization. In essence, it exists to increase efficiency of execution, as individual activities may always run immediately on joining, whereas shared ones may need to pause until other components are ready. Therefore, a lot of the "check and wait" methods needed to synchronize multiple objects are not required and are replaced by null operations. This allows the rest of the framework to treat individual and shared activities identically, as in PEPA.

The Component class defines a `choice` method that takes as argument an array of Activity objects and an array of their respective Rate objects. It returns an integer which corresponds to the position in the array of the Activity which is the fastest ready-to-run branch. The `choice` method is used in a switch statement to select between alternatives. In the case of individual activities there is only one rate, but in shared activities, the overal rate of the action is determined by the slowest rate of the actively participating activities, following PEPA's well-known *apparent rates* method [4]. The participating activities are those members of the subset of the synchronization set that are currently set as the action's *runners*.

In order to discover which actions are ready to run without having to commit, a two-phase process is carried out. The first phase is a "registering of interest" by the component in all the activities which it might choose (i.e. all those in the array). This is done by setting the `ready` flag on the activity to true, indicating to other components that this activity *may* by chosen.

After registering interest in the set of activities, this component *pauses* for “choice-pause” (a constant). Any other `choice` methods that return from their pause may discover that there are now additional (potentially) complete sets in actions they are considering. They may then choose a branch that was previously unavailable. Without such a ready-but-not-locked provision, branches relying on multiple components would be starved—individual activities would always dominate.

Once the thread resumes execution after “choice-pausing”, it will request the locks (in globally-sorted order) of all the actions it is considering joining. It then performs the following steps on each activity it uses for these actions:

1. Set the rate of the activity to be an `almostClone` (same base-rate, different sample) of the one found in the corresponding position in the rate array. This is necessary for cases such as $A = (x, 1).B + (x, 1).C$, where identical rates for the same activities are found in the same choice—this means that two activities can share the same rate object without having identical sleep-times.
2. Use the `getPriority` method to get the determining rate of this activity’s action:
   - For an individual activity, this returns its Rate object.
   - For shared activities, if the action the activity participates in is *not ready* to run, `null` is returned to indicate the branch is unavailable. Otherwise, the method calls `getRate(pathToMyNode)` on the synchronization tree to return the determining rate. An activity is *ready* to run if the *subset* of cooperating *runners* containing this activity are either *locked* (committed and joined), or *ready* (as set by another component’s `choice` method).
3. If the rate returned from `getPriority` is the fastest yet found, this activity is the current *winner*. Note that in the case of equal sleeptimes, one is chosen at random.

If a *winner* is found after checking all activities, the activities that are not chosen have their *ready* flags set back to false, all locks are released, and the position of the *winner* in the activities array will be returned as the result of the `choice` method. Alternatively, if after checking all the activities there are no *ready* actions, all locks are released, the thread will *choice-pause* again, and the process (beginning with the locking) will repeat, until a choice is returned.

The requesting and releasing of locks is necessary to avoid deadlock when there is more than one component considering joining the same action at the same time—locking ensures a *ready* flag is only ever set to true when that activity is available, so that two cooperating components cannot “go opposite ways” in choices. The `Lock` class’s `Lock[] getLockers(Activity[])` method takes a multi-set of activity objects (the same as passed to `choice`) and returns a globally-sorted duplicate-free set of the lock objects of the actions referred to by those activities (each action contains a lock object). This array is passed to the static methods `request(Lock[])` and `release(Lock[])`, which request and release all the locks in order. Global sorting ensures two threads simultaneously requesting different lock sets with shared members will never cause deadlock.

Once a winner is chosen, all the losing sets’ rates are resampled to avoid a branch being unfairly passed over repeatedly. Notice that this includes the resampling of *all* a branch's cooperators—if an activity is passive (PEPA's $\top$ rate), resampling has no effect: the cooperators determine the rate, and they too must be resampled.

Now that the framework for running PEPA models as Java applications has been presented, the translator's functionality and output will be more comprehensible. The translator accepts a slightly reduced subset of the full PEPA grammar, known as *guarded PEPA*. Specifically:

– As justified previously, there is no provision for *hiding* in PEPA2Java.
– The model composition expression may contain only *cooperation* constructs and component identifiers.
– The composition expression is also the *only* place that cooperation constructs may occur.
– *Prefixes*, additional *choices* or grouping structures holding either type of construct are the only valid branches of choice constructs. *Choices* may have any number of branches and may also be nested as long as there is always be an activity to evaluate for each branch. For example,

$$A = (a, 1).A + (b, 2).B + (c, 3).((d, 4).D + (e, 5).E)$$

is a valid construct, but $A = B + (c, 3).C$ is not, regardless of what B defines.

Developing a lexer and parser for the PEPA input was straightforward because the PEPA2Java Translator accepts the same input syntax as the PEPA Workbench [1]. This ensures that the models can be evaluated and refined in the Workbench before generating the PEPA2Java skeletons.

Apart from generating the synchronization set trees, the composition expression is also used to create the Component objects of the system. Each component identifier found in the composition expression will be made into a running instance of a Component-extending class. There may be multiple instances of a single class as is the case in this example:

$$(Client \parallel Client) \underset{\{serve\}}{\bowtie} Server$$

In this example the multiple instances of *Client* will be given names such as `client_i0_Comp` and `client_i1_Comp`, and will be of the `Client_Comp` class.

For the initial behaviour of the Components, equality of identifier is searched for between the component definition in the composition expression and the sequential process definitions. When the sequential process definitions are turned into CompScript objects, the `setStartScript` method is used to set the reference to the initial script that a Component object should execute.

The parser returns a tree made of ProcObj objects for each sequential definition. Turning the tree into an `actions` method of a CompScript object is difficult, as correct nesting is important to ensure proper execution. The component's behaviour trees are traversed to gather a list of all the rates, activities and behaviours (CompScripts) this component uses. These will be defined as fields in the Component class. The followwing model produces the CompScript `A_Script` given in Figure5.

$$A = (m, 1.0).A + (n, 2.0).A + (o, 3.0).((x, 0.5).A + (y, 0.5).A)$$

```
public class A_Script implements CompScript {
   Activity [] ch_Act_0 = {m_Activ, n_Activ, o_Activ};
   Rate [] ch_Rate_0 = {new Rate(1.0), new Rate(2.0), new Rate(3.0)};
   Activity [] ch_Act_1 = {x_Activ, y_Activ};
   Rate [] ch_Rate_1 = {new Rate(0.5), new Rate(0.5)};
    public CompScript actions () {
        switch(choice(ch_Act_0, ch_Rate_0)) {
        case 0: m_Activ.joinAt(new Rate(1.0));
              return a_Script;
        case 1: n_Activ.joinAt(new Rate(2.0));
              return a_Script;
        case 2: o_Activ.joinAt(new Rate(3.0));
              switch(choice(ch_Act_1, ch_Rate_1)) {
              case 0: x_Activ.joinAt(new Rate(0.5));
                    return a_Script;
              case 1: y_Activ.joinAt(new Rate(0.5));
                    return a_Script;
              }
              throw new Error ("Problem with choice: no valid case returned!");
        }
        throw new Error ("Problem with choice: no valid case returned!");
    }
}
```

Fig. 5. CompScript generated from: $A = (m, 1.0).A + (n, 2.0).A + (o, 3.0).((x, 0.5).A + (y, 0.5).A)$

References to sequential process definitions are returned as the next CompScript. *Prefix* constructs become `joinAt(Rate)` method calls. Processing a *Choice* construct is more difficult, because any number of branches is allowed. Some account must be kept of whether each encounter with another *Choice* ProcObj is part of a new choice block, or adding another branch to an existing choice block.

Generating the components, their scripts and defining the system's class instances and synchronization trees by implementing the PEPA system's abstract methods accounts for the majority of the work of the Translator. As a convenience, it has been integrated into the PEPA Workbench, building on previous extensions to its functionality [5, 6]. Additionally, Makefiles are generated to allow models to be easily compiled and run from either within the Workbench or from the command line.

5 Case study

To illustrate the process of translation and execution of a PEPA model in Java, consider the following example of a system with three components: a client that makes requests for information, a server that is the authoritative provider of the requested information, and a proxy that serves the client but may also need to query the server when it cannot fulfil a request itself. Below is the PEPA description of the system:

$$
\begin{aligned}
&a{=}1.0\\
&b{=}2.0\\
&c{=}3.0\\
&\mathit{Client} = (\mathit{cReq}, a).(\mathit{cRep}, \top).\mathit{Client}\\
&\mathit{Proxy} = (\mathit{cReq}, \top).\mathit{Proxy}'\\
&\mathit{Proxy}' = (\mathit{cRep}, b).\mathit{Proxy} + (\mathit{pReq}, a).(\mathit{pRep}, \top).(\mathit{cRep}, b).\mathit{Proxy}\\
&\mathit{Server} = (\mathit{pReq}, \top).(\mathit{pRep}, c).\mathit{Server}\\
&\mathit{Client} \underset{\{\mathit{cReq},\, \mathit{cRep}\}}{\bowtie} \mathit{Proxy} \underset{\{\mathit{pReq},\, \mathit{pRep}\}}{\bowtie} \mathit{Server}
\end{aligned}
$$

Once generated, the PEPA2Java code is compiled and run, either using the Barebones or the Simulator implementations of the PEPA2Java API. The SimWindow is a GUI intended to allow the user to understand the state of the running model, to pause and resume simulation, to interrupt sleeping actions, and to adjust the execution speed and the level of debug messaging. There are three main parts to the SimWindow: the three state tables, the debug message panel and the control panel. The state tables each display information on one of the three main different types of PEPA objects in the model: components, actions and activities. For each component, the current state, the name of the presently executing CompScript, the activity it is participating in and the number of CompScripts run are given. Possible states are, for example, "Waiting for others", "Joined" and "Choosing". For activities, the current state, the rate, the type (shared or individual) and the number of times run are given. Finally, for actions, the state, their determining rate, the status of their synchronization tree (and its nodes) and the number of times run are displayed. The middle portion of the SimWindow gives all system and object debugging messages, along with the name and type of the object from which the message originates. By default, only system status messages are displayed but the level of detail can be modified in the control panel, to the point where a great deal of information can be discovered on exactly what is happening not only in the PEPA model but also what API methods are doing "below the surface." This is useful if the model is behaving incorrectly or unexpectedly. On the bottom row, the control panel makes *play* and *pause* commands available for resuming and pausing the running of the simulation, as well as *interrupt* for prematurely ending long-running actions. Additionally, there are two pull-down lists where simulation speed and the level of debug messaging can be specified.

The simulation produces as its results the percentage of the total system time which each action/ activity incurred. Also computed are the mean sleep time of each run and the number of times each action (shared and internal) was executed. This facility is meant only to provide basic feedback to help the user determine whether the model is working correctly.

If the PEPA2Java code is executing satisfactorily, the next stage is to override the `action` methods for each of the actions in the model, in this case the information request and reply actions. This is a trivially simple example, but because of the synchronization of the activities of the components, the implementor can always be certain that the various components of the system are in a consistent and correct state when they cooperate on shared actions.

6 Conclusions and further work

The principal goal of this work was to devise a method to support the rapid prototyping of high-performance concurrent Java applications. This was accomplished by automating the creation of “skeleton” Java implementations from designs specified in the high-level modelling language, PEPA.

Automating the process means undesirable behaviour such as deadlock will not be introduced. Also, the performance characteristics of the model, tuned to deliver the best results in the Workbench, will be maintained through into implementation. In order to fulfill the goal of the project, two objectives were set.

The first objective was to produce and implement an API for creating and using PEPA-equivalent constructs in Java. The PEPA2Java API has been specified and is implemented by the Barebones and Simulator packages. With the omission of the hiding construct, these can run any PEPA model as a concurrent system.

The second objective was to build an application that could translate PEPA models into Java. The PEPA2Java Translator produces running “skeleton” implementations which use the commands specified in the PEPA2Java API to run as PEPA-equivalent systems.

By automating the process of creating the initial Java implementation from a PEPA model, the result will maintain the performance and behavioural characteristics of the model, providing a sound framework for completing the concurrent application.

Not only do these tools ensure the system characterisics remain true to the high-level model, they remove uncertainty and the potential of introducing human error, whilst also greatly speeding the process of prototyping and implementation.

Acknowledgements: Stephen Gilmore is supported by the DEGAS (Design Environments for Global ApplicationS) project IST-2001-32072 funded by the FET Proactive Initiative on Global Computing.

References

1. S. Gilmore and J. Hillston. The PEPA Workbench: A Tool to Support a Process Algebra-based Approach to Performance Modelling. In *Proceedings of the Seventh International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, number 794 in Lecture Notes in Computer Science, pages 353–368, Vienna, May 1994. Springer-Verlag.
2. S. Gilmore and J. Hillston. Refining internal choice in PEPA models. In R. Pooley and J. Hillston, editors, *Proceedings of the Twelfth UK Performance Engineering Workshop*, pages 49–64, September 1996.
3. S. Gilmore, J. Hillston, and D.R.W. Holton. From SPA models to programs. In M. Ribaudo, editor, *Proceedings of the Fourth Annual Workshop on Process Algebra and Performance Modelling*, pages 179–198, July 1996.
4. J. Hillston. *A Compositional Approach to Performance Modelling*. Cambridge University Press, 1996.
5. J. Hunter. Re-evaluation of the PEPA Workbench. Master’s thesis, The University of Edinburgh, September 1999.
6. F. Stathopoulos. Enhancing the PEPA Workbench with simulation and and experimentation features. Master’s thesis, The University of Edinburgh, September 2001.