14

Compiling Heterarchical Programs by Means of Petri Nets

John B. Evans

Department of Computer Science, University of Hong Kong, Pokfulam Road, Hong Kong

Abstract

We present a method for compiling a program which does not have a conventional tree-like hierarchy of control. Such a structure can arise quite naturally in applications which are strongly user-interactive or contain significant elements of transaction processing, dynamic modelling or process control. Having a heterarchical structure, the application program is capable of being executed simultaneously at an unlimited number of points. Thus a program involving parallel distributed processing, interleaving between processors or multithreading might take advantage of this approach. The encompassing execution strategy is based on the idea of tokens moving through a variant of a Petri net under the control of a firing protocol. We describe an approach to modelling control structures with Petri net semantics and means taken to provide complete reactivation control at run time.

Keyword Codes: D.1.5, D.3.3
Keywords: Object-oriented programming, Language constructs and Features

1. DISCRETE-EVENT SYSTEMS AND PETRI NETS

A discrete-event system is a dynamic system which changes at certain points in time by the instantaneous movement of entities from one state to another. Such a system can be described in terms of entity interactions, any number of which may be simultaneously taking place in parallel. Although the interactions may persist, their initiations and dissolutions are taken to be instantaneous. The system is a discrete-event system; that is, that all change in the system is seen as instantaneous. In many cases this is an assumption which is easy to make: either it is approximately true, or a finite duration can be resolved into a pair of bounding instants. The state of the system as a whole can be thought of the totality of all the states currently taken up by all the component entities. A discrete-event system is thereby a valuable abstract means to specify a wide variety of existing or hypothetical systems.

Discrete-event systems, with their concern for parallel happenings are obvious candidates for some kind of *Petri net* [1] description. In brief, a Petri net is a directed bipartite graph with a net of nodes consisting of a set of *places* (represented by circles) and a set of *transitions* (represented by bars). Being directed, the arcs are of two types: those connecting places to transitions, and those connecting transitions to places. One way of capturing a discrete event system is to associate the *places* with *states* in which the interactions occur and *transitions* with *state-changes*. Entities can be modelled by *tokens* of the net (represented by small discs) which can occupy places which denote the particular interactions being undergone. An association of places with tokens is called a *marking* of the net.

The dynamics of a Petri net are given by an application of the Petri *firing rule* which applies to any transition. If we consider all the places which are associated with a given transition, that is the places which are connected by an arc *from* the place *to* the transition (i.e. the *input places* of the transition), then if each member of the transition's set of input places contain at least one token, the transition can be said to be *enabled* for firing. On *firing* one token is removed from each input place and one token is entered into each *output place*, with complementary meaning. Note that for any transition, its input and output places are not

M. Lee et al. (eds.), *Software Quality and Productivity*

necessarily disjoint. With such an association, the idea of discrete events translates into instantaneous firing.

We propose a variant of the Petri net, the Devnet [2], for which we seek to make the following correspondences to discrete-event systems:

- *place* becomes the state in which the entity may reside for some non-negative duration of time. In applied systems research, the concept of state is closely associated with that of *activity*— the idea being that an entity in a state is actually *doing* something productive. We may maintain this association as long as we are prepared to accept that durations in which entities wait idly for resources, or for some condition to arise, are also counted as 'activities'.
- *transition* becomes a generic event which defines and controls the change from one state to another. Such events can delimit activity durations and initiate entity interactions.
- *token* becomes the individual entity, typed as a member of an entity class. The class definition may include attributes which themselves are typed. Equivalently, we may consider tokens as *objects* representing instances of their respective class. The tokens are coloured according to their class. The instantaneous marking of a Devnet corresponds to the overall configuration of the discrete-event system given by the totality of entities-in-state.
- *arcs* denote the connections between state and event, or between the activities and the events which start and terminate them. More particularly, arcs which are input to a transition indicate an enabling relationship, while arcs which are output from a transition indicate the possible routes of tokens dispersed by the transition's firing. Since arcs refer to the passage of tokens, they are coloured according to the class of entity they transport.
- *firing* then becomes an event occurrence, an activation of an enabled transition where the controls and consequences defined in the event are made manifest on behalf of a specific entity, or collection of entities. Firing involves moving tokens from places input to the transition to places in its output set. In the Devnet we allow some kinds of event to be triggered after a time-delay from their enablemenent.

One particular simulation language, ***Simian*** [3], has an intimate relationship with the Devnet. The compiler constructs a data-structure representation of the Devnet as the temporal aspects the Simian program are being compiled, in order to guide the activation and reactivation of the program at run time. Compilation pragmatics are thus reduced to a protocol of pursuing tokens across transitions, with due regard to token matching, expressed priorities and conflict resolution. By incorporating the Devnet into the object program in this way gives a more coherent and intelligent interpretation of the system described, which in turn enables the design of the Simian language to be *declarative* at an extremely high level.

Given any unmarked Devnet, we can tell which transitions may be in mutual conflict. For any marked Devnet we can tell which transitions are not enabled, and thus the set of transitions whose enablement status should be further investigated. The precise workings of how a Devnet may drive a program is described later. In addition to the typical *Petri interaction*, the Devnet allows *message interaction* in which the state of a remote place may affect (either enable or inhibit) a transition's ability to fire, a *delay interaction* in which an enablement commences a count-down to a later firing for a particular entity in the input place.

The three-way association of entity-event-time (or, in net terms, token-transition-time) is taken as fundamental, forming the *engagement* data-structure, which is the means by which all entity interactions are implemented in Simian. Also present in the data structure is the idea of entities coming together in *collaboration*, as specified in the Petri interaction, for the duration of particular activities. This aspect of object handling has received scant attention from language designers adhering to the OOP paradigm. On transition firing, input tokens can be fused together for collaboration or separated for dispersal. The unfolding of the system dynamic is then seen as a series of making and breaking of engagements as time advances, as prompted by the Devnet structure.

2. THE DEVNET: CONTROL STRUCTURES

There is a large number of ways in which an entity may be guided along a route. We

consider here three general classes of route: choice, iteration and parallel, corresponding to roughly similar features from the control structures of algorithmic programming languages. For the first two classes, a new feature is added to the Devnet, that of a transition which may have *disjunctive* sides. This feature allows the modelling of non-deterministic enablement and dispersal. The separation into upside and downside enables the consideration of the enablement and dispersal properties of the transition separately. Each side can then be broken into a number of segments which represent disjunctive alternatives to the functioning of the respective side. Each segment is considered separately as regards enablement or dispersal.

In the case of the *choice* structure, we have a subnet which is preceded by a transition with a disjunctive downside. This indicates that when the transition fires, a choice is made and one of the two output routes, and thus one of the subnets, is chosen. When either of the subnets is completed, this has the effect of firing of the closing transition which has a disjunctive upside, indicating that only one upside need be enabled for the transition to fire. In this way the Devnet supports the Simian program structure:

```
IF - THEN - ELSE - FI
```

where a choice is taken based on an evaluation of the Boolean expression immediately after the `IF`. With an obvious extension a more general structure is supported:

```
CASE - IN ... OUT - ESAC
```

In a similar way, making use of a transition with disjunctive upsides and downside, the iterative structure

```
FOR .. FROM .. STEP .. TO .. WHILE .. DO - OD
```

can be supported. When handling iteration, one must always bear in mind the possibility that the initial evaluation will lead to not entering the loop body at all, and, for temporal programs, this requires that all loop control logic be associated with a single transition. An infinite loop does not require any control, but is just a linear sequence turned on itself.

3. FIRING PROTOCOL OF THE PRIMATE ALGORITHM

Having an internal representation of the Devnet as a pathfinder for the Simian executive program enables entities to be advanced in a *circumspect* way, taking into account a system-wide perspective. The executive algorithm of Simian, Primate, by making use of the topological properties of the Devnet, will take note of all the conflicts and reactivations which might otherwise be forgotten or swept under the carpet. Moreover, any number of entities can be advanced simultaneously without giving any particular advantage or hindrance to any one of

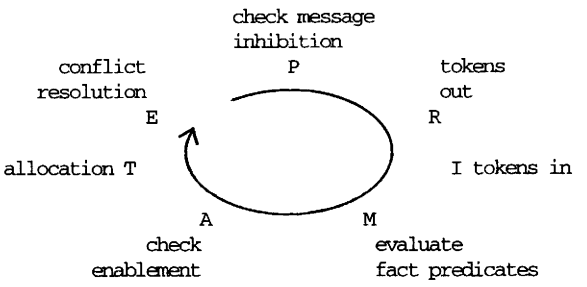

Fig. 1: The Primate algorithm: an outline.
Normal entry is at P, but M, A and E starts are possible. The cycle can be exited after P, M, A, T or E.

them, while maintaining respect of the conditions of priority and conflict resolution, etc. which the simulationist has stipulated.

The first stage (see Fig. 1) is concerned with ascertaining the next event and the particular engagement containing the entity, or set of entities, to be advanced. For ease of description, we will assume here that there is only one engagement, although the algorithm can handle an arbitrary number of coincident engagements. Then the engagement's immediate future is pursued, resulting in a new configuration of the model, giving a new Devnet marking. Finally the consequences of the new marking are analysed to detect whether additional token movement is possible. If so the cycle is continued from the second stage and the cycle is repeated until there is no more token movement possible.

4. CONCLUSIONS AND FURTHER WORK

Although this discussion has been couched mainly in terms of the Simian compiler, a methodology using some variant of Petri nets should be of value in the compiler for any language accepting general heterarchical [4] programs. In pursuing this approach we have departed from three conventional assumptions, *viz*, that the "natural" structure of program control is tree-like and hierarchical; that objects do not form associations, divide and re-divide (especially across class boundaries); and that diagrammatic representations of programs are for necessarily *preliminary* excursions into programming.

We have tried to show that the Devnet diagram, as a qualitative expression of system properties, has some definite benefit as a portrayal of the outline of a system's dynamic. In the form of a data-structure serving as an activator for a heterarchical program it can serve:

- for detecting conflict possibilities;
- as an internal model of event interdependencies;
- to automatically reactivate entities;
- as a basis for driving a back-end animation.

Promising future work on the Devnet might extend its application into the area of the monitoring of control systems involving human controllers, automatic transport, inventory control and in the area of AI planning, both of which are areas with significant dynamic and decision components. By subsuming so much abstract structure into the compiler, we arrive at a higher-level platform upon which to base the design of languages, in the medium of which specific dynamic structures can be definitively *expressed*. The declarative nature of such utterances no longer bears much relation to what was conjured up by the label "programming". Instead a bridge is built between the modes of *specification* and detailed execution. Ultimately the effect of such a close system modelling will amount to a prelude to automation and improvement of the system under study [5].

REFERENCES

1. Reisig, W. *Petri Nets: An Introduction*, Springer-Verlag, Berlin, 1985.

2. Evans J.B. 'The Devnet: a Petri Net for Discrete Event Simulation', (Rozenberg, G., Ed.) *Advances in Petri Nets* 1993, *Lecture Notes in Computer Science,* **674**, (Eds: G. Goos, J. Hartmanis), Springer-Verlag, Berlin, 1993, 91-125.

3. Evans, J.B. *Structures of Discrete Event Simulation: An Introduction to the Engagement Strategy*, Ellis Horwood, Chichester, UK, pp. 279, 1988; ISBN: 0-7458-0103-X.

4. Boden, Margaret *Artificial Intelligence and Natural Man*, Second Edition, MIT Press, London, 1987.

5. Evans, J.B. 'System Synthesis: An Object Petri-Net Methodology', Proceedings of the *European Simulation MultiConference 1994*, Barcelona, June 1-3 1994, 202-206, and University of Hong Kong Department of Computer Science Technical Report TR-93-09, pp. 13.