Effective Object-Oriented Behavioural Modelling with State Nets

Christophe Giraud-Carrier  
Brigham Young University  
Department of Computer Science  
Provo, UT 84602

Abstract

Systems analysis in software engineering is concerned with the modelling of both the static and dynamic aspects of the system under study. However, static modelling seems to have received more attention than behaviour modelling over the years. This is particularly true within the object-oriented paradigm, where behavioural models are generally extensions of finite state machines that relate only loosely to the underlying conceptual basis of object-orientation and may be limited in terms of computational and/or expressive power. The contribution of this paper is two-fold. First, it describes state nets, a novel, object-oriented behavioural model, in which behaviour is expressed by the states of objects, the triggers causing transitions between states and the actions performed by objects in states and transitions. Second, it shows that state nets are effective, in the sense that they are at least as powerful as Turing machines, i.e., they have sufficient computational power to capture arbitrary behaviours.

1. Introduction

Behavioural models play a crucial role in software engineering methodologies as they allow the analyst to go beyond the static aspects of a system and give an account of the dynamics of that system. Although behavioural models are computational models in the traditional sense, their application to systems analysis raises a number of theoretical, as well as pragmatic issues. Like classical formal computational devices, such as the Turing machine, behavioural models must be capable of capturing the dynamics of complex systems. However, because they are often used as means of communications between experts and non-experts, they should also be relatively simple to use and to interpret. Specifically, there are at least three dimensions along which behavioural models can be gauged and compared:

1. Computational power,
2. Expressive power, and
3. Adherence to the underlying conceptual model.

Informally, computational power has to do with how much a model can do (i.e., the range of behaviours it is able to represent effectively). Formal analyses of computational models in terms of their relative computational power have given rise to the well-known Chomsky hierarchy, whose pinnacle is the Turing machine. When a new computational model is introduced, it is often useful to determine where it fits in this global hierarchy. The higher up in the hierarchy, the greater the computational power. From an analyst's
standpoint, a model's computational power is an indication of its usefulness. In fact, one can argue that to be of any real use in software engineering, a behavioural model must be at least as powerful as a Turing machine so as to allow the capture of arbitrary system behaviours.

The notion of expressive, or modelling, power was developed in (Peterson, 1981) and is defined on equivalence classes of computational models. That is, behavioural models are comparable in terms of expressiveness only if they have equivalent computational power. Expressive is concerned with how efficiently (or succinctly) captured behaviours are represented, in terms of the syntactic conventions of the model. The greater the expressiveness, the more economical the representations of behaviours. From an analyst's standpoint, a model's expressive power is an indication of its usability.

All software methodologies have an underlying conceptual model, which is their driving force or the set of concepts upon which they are based. For example, in SA/SD (Yourdon, 1998; Yourdon & Constantine, 1986), a system is viewed as a set of processes and the data that flow between them; in DSSD (Warnier, 1974; Warnier, 1981; Orr, 1977; Orr et al, 1981), a system is viewed as a mapping between input and output structures; in OOA/OOD (Coad & Yourdon, 1990; Booch, 1993), a system is viewed as a set of objects and the relationships between them. All of these methodologies provide models/tools for static and dynamic analysis. In order to avoid undesirable paradigm shifts, all of a methodology's tools should implement naturally that methodology's underlying conceptual model. From an analyst's standpoint, adherence to the underlying conceptual model means seamless integration of all aspects of the system's analysis.

This paper analyses state nets along two of the above dimensions, computational power and adherence to the underlying conceptual model. State nets are the object-oriented behaviour modelling tool used in Object-oriented Systems Analysis (OSA) (Embley et al, 1991). OSA belongs to the family of object-oriented methodologies that have given rise to the widely accepted and now fairly standard Unified Process approach (Booch, 1998; Jacobson, 1999).

Whilst many object-oriented (OO) behavioural models tend to be extensions of finite state machines that relate only loosely to the underlying conceptual basis of object-orientation and are often limited in terms of computational and/or expressive power, state nets are both effective and efficient, as well as true to the OO philosophy. State nets capture behaviour by representing the states of objects, the conditions and events causing transitions between states, and the actions performed by objects in states and transitions. Both intra-object concurrency and inter-object concurrency are embedded in the state net model.

This paper provides a general description of state nets and formally shows that state nets are at least as powerful as Turing machines. The proof is direct and constructive. It is shown that an appropriately designed state net can simulate the Turing-equivalent Shepherdson and Sturgis's (1963) register machine. This result serves as a foundation for the comparison of state nets with other computationally equivalent behavioural models on the basis of their relative expressive power. For example, state nets have been shown to be more expressive than Petri nets (Giraud-Carrier et al, 1993).

---

1The original proof in (Giraud-Carrier, 1991) was found to have several deficiencies that this paper addresses. Many needed details and clarifications have also been added.
State nets extend finite state machines to produce a flexible behavioural model that gives a software analyst a way of effectively specifying both sequential and parallel systems in an object-oriented context, where both intra-object concurrency and inter-object concurrency are supported.

A state net is labelled with the name of an object class and is effectively a template for the behaviour of the object instances of that object class. Figure 1 is an example of a state net for a very simple object class called *Bank Account*.

**Bank Account**

![State Net Diagram](image)

Figure 1 - State Net for the Object Class *Bank Account*

The “memory” of a state net is stored in an instance of an Object-Relationship Model (ORM), which is another subpart of the OSA model. ORMs are semantic data models, extensions of the well-known Entity-Relationship Model, whose diagramming conventions and semantics are formally defined in (Embley et al, 1991).

The ORM diagram for the object class Bank Account is depicted in Figure 2. Boxes represent object classes and are labelled with an object class name. Lines connecting boxes represent relationship sets that serve as templates for the relationships amongst objects of the connected object classes. The numbers at each end of a connection represent cardinality constraints. These constraints are of the form $min:max$ and specify the minimum and maximum numbers of times that an object in the connected object class participates in relationships of the associated relationship set. The ORM diagram
of Figure 2 consists of three object classes, `Bank Account#`, `Bank Account`, and `Balance`, and two relationship sets `Bank Account# identifies Bank Account` and `Bank Account has Balance`. The cardinality constraints constrain a bank account to have exactly one bank account number and exactly one balance, constrain a bank account number to identify exactly one bank account, and constrain a balance to be associated with at least one bank account.

Informally, an ORM instance is to a state net what a tape is to a Turing machine. Just as a Turing machine needs to know which character its tape head is pointing to, a state net needs to know which objects currently exist and which relationships among the objects currently hold in its associated ORM instance. The triggers of the transitions of the state net are then interpreted as queries to the object classes and relationship sets of the associated ORM instance, while the actions are updates to object classes and relationship sets. The presence (respectively, absence) of certain objects and relationships in the ORM instance causes certain triggers to be true and others to be false, thus resulting in certain transitions firing while others remain inactive. Hence, it is the combination of a state net and its associated ORM instance that makes up the actual computational model. The state net in Figure 1 makes explicit use of the information stored in the ORM of Figure 2.

We now return to Figure 1. The rounded boxes are states. Each state has a name, placed inside of the rounded box. In Figure 1, there are 2 states, labelled `current` and `overdrawn`. The rectangular boxes are transitions. Transitions may be labelled by a number in square brackets, above the upper left corner of the box, as in Figure 1. The top half of a transition box contains its trigger. A trigger consists of either a condition that is true or an event that happens. Formally, conditions are Boolean expressions over the object classes and relationship sets of the model instance (i.e., over the facts stored in the associated ORM instance/database). Events are instantaneous and the special prefix `@` is used to distinguish them from conditions. The bottom part of a transition box contains its action, if any. Actions are operations that take place when the transition fires and before the object changes state. As with conditions, actions are simple database operations, namely add/remove/modify an object/relationship, applied to the model instance. In Figure 1, all transitions, with the exception of transitions [1] and [5] have actions. In transition [4], for example, both the trigger and the action reference the Balance object class in the ORM diagram of Figure 2, and in transition [2] the action references and updates the associated object in the Balance object class. Transition [1] is a special case. It is an initial transition, with neither condition nor action. Implicitly, an initial transition always causes the creation of an object, together with its ID.²

A state is either on or off. A transition is in exactly one of the inactive, ready, starting, executing, or finishing states at any time. When an object is instantiated/created, its associated state net is also instantiated with all states off and all transitions inactive. A state is turned on when a transition leading into it fires.

Arrows (directed arcs) generally join one or more states to a single transition. Arrows pointing to transitions are called in-arrows, while arrows pointing away from transitions are called out-arrows. A transition may have 0 or more in-arrows and/or out-arrows. An in-arrow always has a single head and an out-arrow always has a single tail. A bar across an arrow marks an exception. Exceptions are events or conditions that are not part of the normal system's behaviour. The exception in Figure 1 captures the undesirable case of

²OSA does provide a complete formalism of this and other aspects in the form of a meta-model and its associated model-theoretic interpretation. Its presentation is outside the scope of this paper. The reader is referred to (Clyde, 1993) for details.
overdrafts. Exceptions out of transitions have precedence over all other normal out-arrows. Hence, in Figure 1, if a withdrawal causes the account's balance to become negative, then the account enters the Overdrawn state, rather than returning to the Current state, and stays in the Overdrawn state until a sufficient deposit has been made and the applicable fee has been paid.

Arrows are of three kinds and support both sequential and concurrent behaviours, as follows.

- **Single-head/single-tail arrow:** Sequential, single-path behaviour. The object moves from a single state to a single transition or from a single transition to a single state.
- **Multiple-heads/single-tail:** Concurrent, multiple-paths behaviour. Each state pointing to by an arrowhead is turned on upon completion of the transition. This corresponds loosely to the Unix fork operation.
- **Single-head/multiple-tails:** Concurrent, multiple-paths behaviour. All of the states at the tails of the arrow must be on to enable the transition. This corresponds loosely to the Unix wait() system call, where the transition's trigger is the condition on which all states, once executed, must wait to proceed.

In addition to intra-object concurrency, where one object is in more than one state or transition at a time, state nets also support inter-object concurrency. Since each object in an object class has its own "executable" copy of the state net for that object class, several objects may be in various states and transitions at the same time. Random and constrained non-deterministic behaviour may also be captured in state nets through transitions having more than one in-arrow, transitions having more than one out-arrow and state overlap between several transitions' in-arrows. Real-time constraints and abstraction mechanisms (e.g., high-level states) can also be captured by the model. Details are in (Embley et al, 1991).

States connected to an in-arrow form a **prior-state conjunction**, while states connected to an out-arrow form a **subsequent-state conjunction**. A prior (respectively, subsequent) state conjunction is on if and only if all of its states are on. In order to explain the behaviour (i.e., execution) of state nets, the following functions and predicates are also needed.

- **T_STATE(t):** the state of transition t (i.e., inactive, ready, starting, executing, or finishing).
- **S_STATE(s):** the state of state s (i.e., on or off).
- **ENABLED(t):** true if and only if, either transition t has no prior-state conjunctions, or at least one of t's prior-state conjunctions is on.
- **TRIGGER(t):** the Boolean value of transition t's trigger.
- **ACTION(t):** the action associated with transition t if any; else, NULL.
- **PSC(t):** the set of transition t's prior-state conjunctions.
- **SSC(t):** the set of transition t's subsequent-state conjunctions.
- **SELECTED(t):** the selected prior-state conjunction of transition t, which may be NULL.
- **READY_CONFLICT(t):** true if and only if transition t is ready and there exists another transition t' that is ready and at least one on
prior-state conjunction of \( t' \) contains a state that is also in an on prior-state conjunction of \( t \).

**STARTING_CONFLICT(\( t \))**: true if and only if \( t \) is \emph{ready} and there exists another transition \( t' \) that is \emph{starting} and SELECTED(\( t' \)) contains a state that is also in an on prior-state conjunction of \( t \).

**SELECTABLE(\( t \))**: true if and only if READY_CONFLICT(\( t \)) and \( \neg \text{STARTING_CONFLICT}(t) \).

The execution of a copy \( C \) of some state net \( SN \) can now be defined by the algorithm of Figure 3, which independently executes for every transition \( t \) in \( C \). (Note, if \( S \) is a set, then \(|S|\) denotes the cardinality of \( S \)).

```
WHEN T_STATE(\( t \)=inactive AND ENABLED(\( t \) AND TRIGGER(\( t \)) DO
   T_STATE(\( t \)=ready
WHEN T_STATE(\( t \)=ready AND \( \not\) ENABLED(\( t \) OR \( \not\) TRIGGER(\( t \)) DO
   T_STATE(\( t \)=inactive
WHEN T_STATE(\( t \)=ready AND \( \not\) READY_CONFLICT(\( t \)) AND \( \not\) STARTING_CONFLICT(\( t \)) DO
   IF |PSC(\( t \)|)=0 THEN
      SELECTED(\( t \)=NULL
   ELSE
      SELECTED(\( t \)=some non-deterministically chosen element of PSC(\( t \))
   T_STATE(\( t \)=starting
WHEN T_STATE(\( t \)=starting DO
   IF SELECTED(\( t \)=NULL THEN
      FORALL \( s \) \in SELECTED(\( t \)) DO
         S_STATE(\( s \)=off
      T_STATE(\( t \)=executing
WHEN T_STATE(\( t \)=executing DO
   IF ACTION(\( t \)=NULL THEN
      perform ACTION(\( t \))
      T_STATE(\( t \)=finishing
WHEN T_STATE(\( t \)=finishing DO
   IF |SSC(\( t \)|)=0 THEN
      \( q \)=some non-deterministically chosen element of SSC(\( t \)
      FORALL \( s \) \in q DO
         S_STATE(\( s \)=on
      T_STATE(\( t \)=inactive
   IF |{\( t \) : SELECTABLE(\( t \)}|>1 THEN
      \( t' \)=some non-deterministically chosen element of \{\( t \) : SELECTABLE(\( t \)}
      \( p \)=some non-deterministically chosen on element of PSC(\( t' \)
      SELECTED(\( t \)=p
      T_STATE(\( t \)=starting
```

**Figure 3 - State Net Execution Algorithm**

Exceptions are handled as such. If the trigger of any exception becomes true, the normal execution is interrupted, the state of all transitions is reset to inactive and the state of the state pointing to by the exception arrow is set to on. Execution then resumes as per Figure 3.

If at any point in time, T_STATE(\( t \)=inactive for all transitions, \( t \), of all state nets in the system, and there does not exist a transition \( t' \) such that ENABLED(\( t' \)) and TRIGGER(\( t' \)) are true, then the state net system is said to be deadlocked.
3. Shepherdson and Sturgis's Register Machine

Shepherdson and Sturgis's (1963) register machine arose as a result of their study of the computability of recursive functions. Since it operates only on a finite number of registers, I shall refer to their machine simply as the Bounded Register Machine (BRM). The BRM consists of a finite number, $R$, of registers and it is orchestrated exclusively by the three types of instructions given in Table 1. I denote by $<r>$ the value stored in register $r$, where $1 \leq r \leq R$.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P(r)$</td>
<td>Add 1 to $&lt;r&gt;$</td>
</tr>
<tr>
<td>$D(r)$</td>
<td>Subtract 1 from $&lt;r&gt;$ if $&lt;r&gt; \neq 0$</td>
</tr>
<tr>
<td>$J(r)[k]$</td>
<td>Jump to $k$ if $&lt;r&gt; \neq 0$</td>
</tr>
</tbody>
</table>

Table 1: The BRM's Instruction Set

The programs executed by the BRM consist simply of finite sequences of labelled instructions drawn from the BRM's instruction set. Without loss of generality, the instructions of a program of length $N$ can be assumed to be labelled from 1 to $N$. When given a program to execute, the BRM always starts with the first instruction (i.e., labelled 1). By default, execution proceeds sequentially, from instruction $i$ to instruction $i+1$. However, the $J(r)[k]$ instruction provides a mechanism to alter this sequential flow of execution by conditionally transferring control to instruction $k$ (where in principle, $1 \leq k \leq N$). A program's execution terminates when control is transferred to a position beyond the last instruction of the program. This happens after execution of instruction $N$ or when $k > N$ in an executable jump instruction. The state of the BRM is defined completely by the contents of its $R$ registers and a program counter (i.e., pointer to the next instruction to execute).

Shepherdson and Sturgis (1963) proved that the BRM computes all partial recursive functions. Their main theorem, that the BRM is Turing-equivalent, follows immediately from the following two propositions.

1. The concept of computable as applied to a function of natural numbers is correctly identified with the concept of partial recursive.
2. All partial recursive functions can be computed by Turing machines.

This result establishes the BRM as a powerful computational device with yet relatively simple semantics. Consequently, any new computational model $M$ may be proved to be at least as powerful as a Turing machine by constructing an instance of $M$ that faithfully simulates the behaviour of the BRM. This procedure is used in section 4, where $M$ is a state net.

4. State Nets Are Turing-Equivalent

In this section, I show how the BRM can be simulated exactly by a state net. As mentioned in section 3, the successful construction of such a state net is sufficient to prove that state nets have at least as much computational power as BRMs and, consequently, are at least as powerful as Turing machines.
**Theorem**

State nets are at least as powerful as Turing machines.

**Proof**

The proof consists of the construction of a state net, $SN_{BRM}$, that simulates exactly the BRM. I first describe how $SN_{BRM}$ is constructed and then show that it does indeed faithfully capture the behaviour of the BRM. A formal account of the semantics of $SN_{BRM}$ is given.

$SN_{BRM}$ serves as a template for the behaviour or execution of all instances of the class of BRMs. Let $PC$ be a program counter that tracks the non-negative integer value of the label of the current instruction being executed by the BRM. I say that $PC$ points to the instruction labelled $k$ whenever $<PC>=k$. For example, if $<PC>=3$ and instruction 3 is $D(r)$, then $PC$ points to $D(r)$.

$SN_{BRM}$ is depicted in Figure 4. The state net consists of a two states, *Ready to execute next instruction* and *Finished* and seven transitions. With the exception of the initial and final transitions, each transition corresponds to one of the types of instructions of the BRM and the associated action reflects the changes of states of the BRM. Double-headed arrows are simply a shorthand for two arrows in each direction.

---

Transition [1] is the initial transition. Its associated action initialises the program counter, $PC$, to point to the first instruction of the program. Transition [2] implements instructions of type $P(r)$. Its trigger guarantees that it only fires if the current instruction is of type $P(r)$ and its associated action ensures that the value stored in register $r$ is properly incremented, as well as the value stored in $PC$ (i.e., $PC$ now points to the next instruction). Transitions [3] and [4] implement instructions of type $D(r)$. Two transitions are needed as the outcome depends on the current value stored in register $r$. Transition [3] ensures that if the value stored in register $r$ is 0, then the value of register $r$ remains

---

3The BRM executes only one program at any given time so that $PC$ is well defined.
unchanged and the program proceeds to the next instruction. Transition [4], on the other hand, handles the case where the value stored in register \( r \) is not zero. In this case, the value stored in register \( r \) is first decremented and the program then proceeds to the next instruction. Finally, transitions [5] and [6] implement instructions of type \( J(r)[l] \). Again, two transitions are needed as the outcome depends on the value stored in register \( r \). Transition [5] guarantees that if the value stored in register \( r \) is not zero, then \( PC \) is made to point to the instruction whose label is \( l \) and execution normally proceeds from there. Transition [6] handles the situation where the value stored in register \( r \) is zero. In this case, the program normally proceeds to the next instruction.

Note that in order to handle program termination adequately, all transitions (except for the initial one) have the same exception, which fires when \( PC \) points beyond the last instruction of the program. Here, \( N \) denotes the number of instructions in the program. The semantics (see Section 3) ensure that the trigger of an exception is evaluated before the triggers of any other possible transitions. This is particularly important in this context as it may not be possible to evaluate some of the other triggers if \( PC \) points outside the range of the program. The exception takes the program to the Finished state and a final transition, with neither condition nor action, ensures that the state net is “turned off” when the program stops.

A simple ORM diagram associated with \( SN_{BRM} \) is in Figure 5. The ORM consists of ten object classes, four binary relationship sets, two generalisation/specialisation, one association, and one aggregation.

![Figure 5 - ORM Associated with SN_{BRM}](image)

The object class Register is finite and contains exactly \( R \) objects, all of which are registers, each holding exactly one non-negative integer value. The object class Instruction is a generalisation. The open triangle is shorthand for the is-a relationship set. Instruction has three specialisations or subclasses, one for each BRM instruction type, namely \( P()\)-Instruction, \( D()\)-Instruction and \( J()[]\)-Instruction. The set of instances of a subclass forms a subset of the set of instances of its superclass. The parametrisation of the BRM's instructions is captured in the ORM by the two binary relationship sets,
Instruction references Register and J[[]]-Instruction references Label. The participation constraints on the relationship sets are such that all instructions reference exactly one of the $R$ registers and only the instructions of type J[[]] also reference exactly one label. Registers and labels may be referenced in an arbitrary number of instructions (including none at all). The object class Labelled Instruction is an aggregate. The filled-in triangle is shorthand for the is-part-of (reciprocally, is-made-up-of) relationship set. Hence, a labelled instruction consists of exactly one label and exactly one instruction. Each label is also had by exactly one labelled instruction but the same instruction may have several labels (i.e., same instruction appearing in different places in the program). As the BRM never runs more than one program at a time, Program and PC are singleton object classes. The two notations are semantically equivalent. The class Program is an association. The star notation is shorthand for the is-member-of (reciprocally, is-set-of) relationship set. Hence, a program is a (possibly empty) set of labelled instructions. The general constraint in the top right-hand corner of the diagram further restricts that set by stating that a program's instructions are organised in a sequence from 1 to $N$. The object class Label is a specialisation of the object class Non-Negative Integer Value.

The procedural semantics of $SN_{BRM}$ are captured by the semantics of the well-defined loop structure of Figure 6. The initial transition [1] corresponds to the initialisation statement before the loop (i.e., DO statement). The final transition [2] corresponds to the exit condition of the loop. Every other transition maps trivially to one of the cases within the body of the CASE OF statement.

\begin{verbatim}
BEGIN
  <PC>←1
  DO
    CASE OF
      PC points to P[[]]:
        <r>←<r>+1
      PC points to D[[]] AND <r>=0:
        <PC>←<PC>+1
      PC points to D[[]] AND <r>≠0:
        <r>←<r>-1
        <PC>←<PC>+1
      PC points to J[[]] AND <r>=0:
        <PC>←<PC>+1
      PC points to J[[]] AND <r>≠0:
        <PC>←<PC>+1
      UNTIL (<PC> > N)
  END
\end{verbatim}

Figure 6 - Procedural Semantics of $SN_{BRM}$

The declarative semantics of $SN_{BRM}$ are captured by queries and updates to the ORM of Figure 5. ORM diagrams can be viewed as a form of extended entity-relationship diagrams. In particular, the ORM diagram of Figure 5 can be transformed into an equivalent instance of the Extended Conceptual Entity Relationship (ECER) model proposed in (Czejdo et al, 1988; Czejdo et al, 1991)\(^4\). It follows that relation schemes can be constructed and that the triggers and actions of the transitions of $SN_{BRM}$ can be formalised using tuple calculus notation. To simplify the expressions, the object classes and relationship sets are abbreviated as follows.

<table>
<thead>
<tr>
<th>Object Class</th>
<th>Abbreviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>P: Program</td>
<td>C: PC</td>
</tr>
</tbody>
</table>

\(^4\)The transformation from ORM instance to ECER instance is not always possible, however, as the two models are not equivalent.
LI: Labelled Instruction
PI: P()-Instruction
JI: J()-Instruction
N: Non-Negative Integer Value
PsLI: Program is a set of Labelled Instruction
IrR: Instruction references Register
CpLI: PC points to Labelled Instruction
JIrL: J()-Instruction references Label
RhN: Register holds Non-Negative Integer Value
LpLI: Label is part of Labelled Instruction
IpLI: Instruction is part of Labelled Instruction

Next, the following may be defined:

\[ L_c = \{ x[L]: \exists y[L \in LpLI \land y[L] = x[L] \land \exists z[C \land L \in LpLI \land z[L] = y[L]) \} \]

The label of the labelled instruction pointed to by PC

\[ I_c = \{ x[I]: \exists y[I \in IpLI \land y[I] = x[I] \land \exists z[C \land L \in IpLI \land z[L] = y[L]) \} \]

The instruction pointed to by PC

\[ R_c = \{ x[N]: \exists y[R \in RhN \land y[N] = x[N] \land \exists z[I \in IrR \land z[R] = y[R] \land \exists t[L \in LpLI \land t[L] = z[I] \land \exists v[C \land L \in LpLI \land v[L] = t[L]]) \} \]

The value stored in the register referenced by the labelled instruction pointed to by PC

\[ J_c = \{ x[L]: \exists y[JI \in JIrL \land y[L] = x[L] \land \exists z[I \in IpLI \land z[I] = y[JI] \land \exists t[C \land L \in LpLI \land t[L] = z[L]]) \} \]

The label referenced by the labelled instruction of type J() pointed to by PC

Finally, the triggers of the transitions of \( SN_{BRM} \) may be formally expressed as follows:

Transition [1]: Not applicable
Transition [2]: \( I_c \in PI \)
Transition [3]: \( I_c \in DI \land R_c = 0 \)
Transition [4]: \( I_c \in DI \land R_c \neq 0 \)
Transition [5]: \( I_c \in JI \land R_c \neq 0 \)
Transition [6]: \( I_c \in JI \land R_c = 0 \)
Exception: \( L_c > N \)

The actions in \( SN_{BRM} \) that involve \(<PC>\) correspond to updates to CpLI while those involving \(<r>\) correspond to updates to RhN. For example, the action of transition [5] consists of setting the value of PC to that of \( J_c \). All of the necessary updates are well-defined, so the actions of \( SN_{BRM} \) are also well-defined. Consequently, the ORM of Figure 5 accurately acts as storage for \( SN_{BRM} \).

From the above construction and the associated formal description, it is clear that \( SN_{BRM} \) captures exactly the behaviour of the BRM. Every instruction of the BRM is effectively simulated by \( SN_{BRM} \). The only actions performed by \( SN_{BRM} \) appear in the associated actions of its transitions and there are no other actions in \( SN_{BRM} \) than those corresponding strictly to the implementation of the instructions of the BRM\(^5\). Since the BRM is Turing-equivalent, it follows immediately that state nets are at least as powerful as Turing machines.

---

\(^5\)The action of transition [1] is not added. It is implicit in the description of the behaviour of the BRM.
5. Conclusion

An object-oriented behavioural modelling tool, called state nets, has been described. The general syntax and semantics of state nets have been exposed. By constructing a state net that exactly simulates the behaviour of Shepherdson and Sturgis's Turing-equivalent register machine, it is also formally established that state nets are at least as powerful as Turing machines. It follows that state nets have sufficient computational power to capture arbitrary behaviours, which is not the case, for instance, for state charts (Harel, 1987).

The significance of this result is more than theoretical. Although having at least as much computational power as a Turing machine is in some respect a prerequisite for a behavioural model’s general acceptance and use in software engineering, it is clearly not an end in itself. Many computational models, including the Turing machine, have been proposed over the years, yet most of them have been of little value to software engineers because of their complexity of use or the awkwardness of their representation. The goal of researchers in software engineering is the production of behavioural models that are not only computationally powerful but also, and indeed more importantly so, expressively powerful. That is, they must be flexible, easy to use for the practitioners and relatively compact in their representation.

Though a full discussion of the expressiveness of state nets is beyond the scope of this paper, the brief treatment given here does provide some insight into the modelling power offered by state nets. A comparison with Petri nets was reported in (Giraud-Carrier et al, 1993).

References


