On state-oriented vs. event-oriented thinking in formal behavioural specifications

T. Bolognesi
CNR – ISTI – Pisa
t.bolognesi@isti.cnr.it

Abstract. We introduce a simple conceptual framework for assessing a number of well known formal specification techniques w.r.t. their ability to model state-oriented and/or event-oriented aspects of system behaviour. By attributing a-priori equal importance to the notions of event and state, by explicitly recognizing the two derived, *very* fundamental ways of thinking about system behaviours, and by assessing the bias of existing formal methods towards one or the other, one can make more conscious decisions and selections in the upper phases of software development, that is, in requirements elicitation and analysis, in the construction of abstract system models, and in the choice of formal languages for high- and low-level design. In particular, we assess the recently introduced model of Abstract State Processes, and the design choices behind its definition, in light of the introduced state-event framework.

Keywords – D.2.1 Requirements/Specifications, D.2.2 Design Tools and Techniques --- Formal methods, Process algebra, ASM, B, CCS, CSP, High-level Petri nets, LOTOS, TLA.

Contents

1. Introduction

In this paper we refer to the problem of formally specifying complex (software and/or hardware) concurrent, distributed, reactive systems. In the early phases of system development *abstract models* of the system are built, possibly by means of formal specifications, in order to clarify system requirements. Among the several formal or semi-formal specification techniques available for building abstract system models, we are interested here on those that address explicitly system *behaviour*. Thus we exclude, for example, Data Flow diagrams, Entity Relation diagrams and (UML-) Class Diagrams, and we refer to formal specification techniques that allow one to express dynamic aspects: which *events* happen when the system operates, and how does the system *state* evolve? We shall use the terms *behavioural specification* and *behavioural specification language.*

In [AL93] Abadi and Lamport write:

> The popular approaches to specification are based on either states or actions. In a state-based approach, an execution of a system is viewed as a sequence of states, where a state is an assignment of values to some set of components. An action-based approach views an execution as a sequence of actions. These different approaches are, in some sense, equivalent. An action can be modeled as a state change, and a state can be modeled as an equivalence class of sequences of actions. However, the two approaches have traditionally taken very different formal directions. State based approaches are often rooted in logic, a specification being a formula in some logical system. Action-based approaches have tended to use algebra, a specification being an object that is manipulated algebraically. Milner's CCS is the classic example of an algebraic formalism […].
> State-based and action-based approaches also tend to differ in practice. […]

The excerpt above represents one of the main motivations for our work. Our objective is to further investigate the issue of state-based vs. action-based specification, in the context of the early phases of (software) system development, namely those that involve the initial brainstorming about system functionalities, the elicitation and analysis of system requirements, the construction of abstract system models, and the choice of formal languages for high- and low-level design. We therefore focus on the expressive flexibility of languages – on their appeal to human intuition – rather than on analytical power, or support to formal verification tasks.

Some software engineers seem to believe that the first, crucial step in system development is the identification of the right global state structure. Others seem to prefer reasoning in terms of external viewpoints: they first describe the system interactions with the environment as patterns of events, without immediately worrying about the state. But, is it really possible, and convenient, to hit the extremes of this spectrum, and have a *purely* state-oriented or *purely* event-oriented formal specification? And how do existing formal specification languages support an integration of these two descriptive modes? If there are multiple ways to do it, where are the advantages and disadvantages?

Consider specification approaches such as ASM [G93, BS03], B [A96], CCS [M80], CSP [H85], High Level Petri Nets [JR91], Statecharts [H87], UML-State Machines [BRJ99], TLA [L03], Z [S89]. Most people would agree in regarding some of them as closer to state-oriented thinking (e.g. ASM), and some others as closer to event-oriented thinking (e.g. CSP); but what does it really mean for a formal specification language to be *state oriented* or *event oriented*? Can we provide a simple conceptual framework or grid where existing formal specification languages can be positioned and compared with one another based on their bias towards states of events?

The purpose of this paper is to shed some light on these questions. Frequently the selection of a specification method – a choice which may influence in various ways the subsequent phases of system development -- is driven by non-technical reasons such as tradition, corporate policy, or even dogma. Our simple state-event framework is meant to contribute in supporting more conscious and perhaps more successful choices, based on primitive but technical criteria.

In Section 2 we introduce a first, elementary version of the conceptual framework, in form of a simple diagram, and we discuss two instances of it (two existing specification methods) that actualize in different ways its constituent elements. In Section 3 we refine the framework, based on the need to handle multiple event types, multiple fragments of the global state, and multiple specification 'chunks'. These multiplicities lead us to naturally distinguish among *five* basic *constraint types*, corresponding to as many behavioural specification paradigms.

In Section 4 we discuss the two most simple constraint types: *invariants* and *pure event-event constraints*. Section 5 is devoted to the more advanced specification paradigms, that we call *disjoint-events/shared-variables* and *shared-events/disjoint-variables*; two instances of these paradigms are discussed. In Section 6 we assess the recently introduced model of *Abstract State Processes*, and the design choices behind its definition, in light of the discussed state-event framework. In the concluding Section 7 we provide a list of research topics related to the conceptual vehicle introduced in the paper.

For space reasons, we cannot provide introductions to the several formal languages considered in the paper, and we can only provide small examples of formal specifications that reflect (some of) the discussed paradigms. We defer more substantial coverage of these components to a longer version of this paper.

2. Basic framework

Figure 1.a illustrates a well known concept in electrical engineering: the synchronous sequential circuit. At each clock step the circuit, based on the current state *s* and the input sample *in,* produces the output sample *out* and a new state, say s'. The outputs are defined by a combinational (stateless) circuit, that represents the 'logics' of the system. The way most behavioural specification languages work can be ultimately described by the diagram in Figure 1.b, which proposes in a new setting the basic elements of Figure 1.a. This diagram implies that *states* and *events* are primitive concepts of equal importance in behavioural specification.

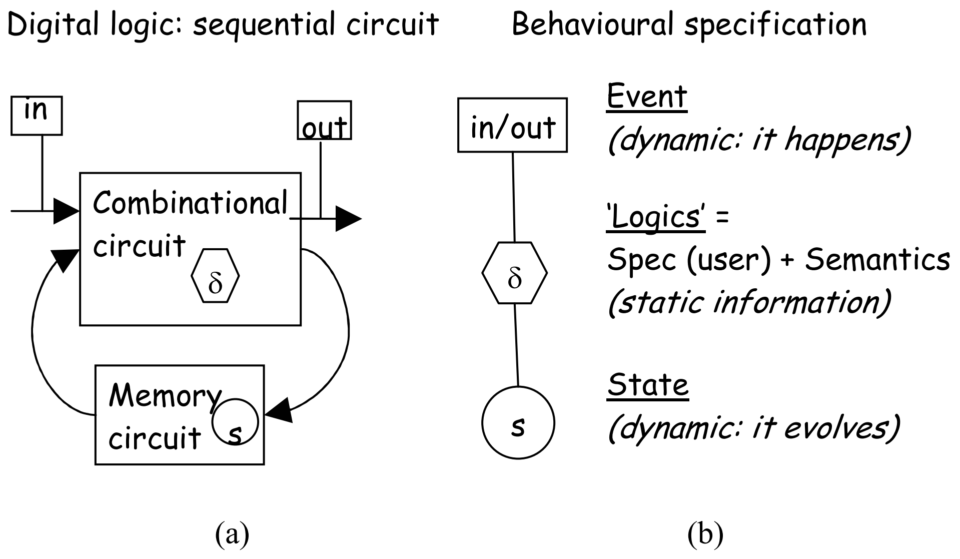

Figure 1 – Basic framework for behavioural specification languages

In Figure 1.b:

- Circles are intended to represent state variables or, more generally, portions of a global state structure.
- Rectangles represent events, which may be structured, and may involve data provided from, and/or offered to the environment. Event occurrences are instantaneous; they may depend on the current state value, and affect it according to some 'logics'.
- Hexagons represent the 'logics' that connects the state and the events, that is, the actual formal specification of the system, be it a piece of text or a diagram; this specification would be meaningless without the semantic rules of the language, which are therefore also considered, perhaps implicitly, as part of the 'logics'. We adopt the term *constraint* for this element, since it enforces some correlation between event occurrences and values on one hand, and state values on the other.

The system specification and the language semantics are the *static* elements of the picture; the state and the event are the *dynamic* ones. State variables tend to persist, but events make them change. Events disappear. One can make future use of the value of an event only by recording it somewhere in the state, as the event happens. Apart from this difference, states and events may well be of the same type (e.g., tuples of natural numbers). In Figure 2 the state structure is represented by a dynamic set of state variables that may change their values (colours) as events occur.

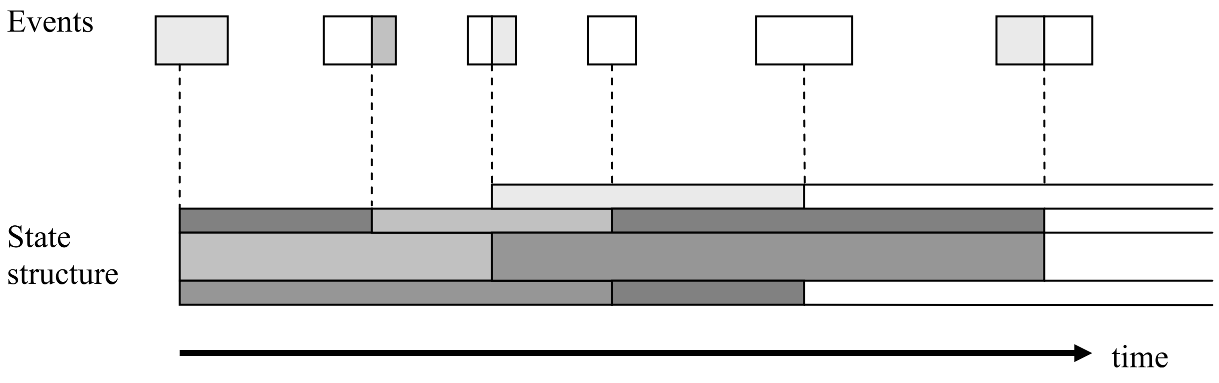

Figure 2 – Events and state changes

2.1 Instances

We briefly consider here two instances of the basic framework of Figure 1.b. The purpose is to emphasize how formal specification languages may differ in the way they conceive and represent events, states and constraints.

Predicate/Transition nets

A Predicate/Transition net [JR91] is a bipartite graph where nodes can be *places* (circles) or *transitions* (boxes), and arcs are labelled by multisets of terms representing *token* values. A finite set of tokens distibuted over one or more places represents the *initial marking*. Transitions are labelled by first order formulae. A transition *T* with label *P* can fire when its input places contain tokens with the values as specified by its incoming arc's labels, and *P* can be satisfied; the effect is to inject tokens with values defined by *T*'s outgoing arc labels into *T*'s output places.

The correspondence between the diagram of Figure 1.b and Predicate/Transition nets is as follows (see Figure 3):

- An *event* is the firing of a *transition*. Some of the variables appearing in *P* can be readily interpreted as event parameters, whose values can be provided or used by an external, unspecified observer.
- The *state* is the current *marking*.
- The system *specification* is the *net* itself, which includes the initial marking
- The operational *semantics* is the *transition firing game*

Full LOTOS

Full LOTOS [BB87] is a process algebraic language. System behaviours are described by behaviour expressions, built by behavioural operators, and by value expressions used for denoting data values. A system is conceived as a dynamic set of processes that interact with one another and with their environment by synchronising (rendez-vous) at so called *gates*, and by possibly exchanging data values.

The correspondence between the diagram of Figure 1.b and Full LOTOS and is as follows (see Figure 3):

- An *event* is … a LOTOS event, consisting of a *gate name* and, possibly, *a tuple of parameter values*.
- The *state* is the current *behaviour expression*: this evolves as the events occur, and during evolution its data variables are substituted for actual values that play the role of *state variables*, although LOTOS is not an imperative language (LOTOS exploits single-use, 'logical' variables, that cannot be re-assigned values). A behaviour expressions acts as a complex state which implicitly defines all the transitions immediately possible from for that behaviour, that is (i) all the events that may occur immediately, and (ii) the residual behaviour expression obtained after each of these events.
- The *system specification* is the LOTOS text.
- The operational *semantics* is the set of axioms and inference rules of the LOTOS SOS (Structural Operational Semantics), that formally define the transition relation.

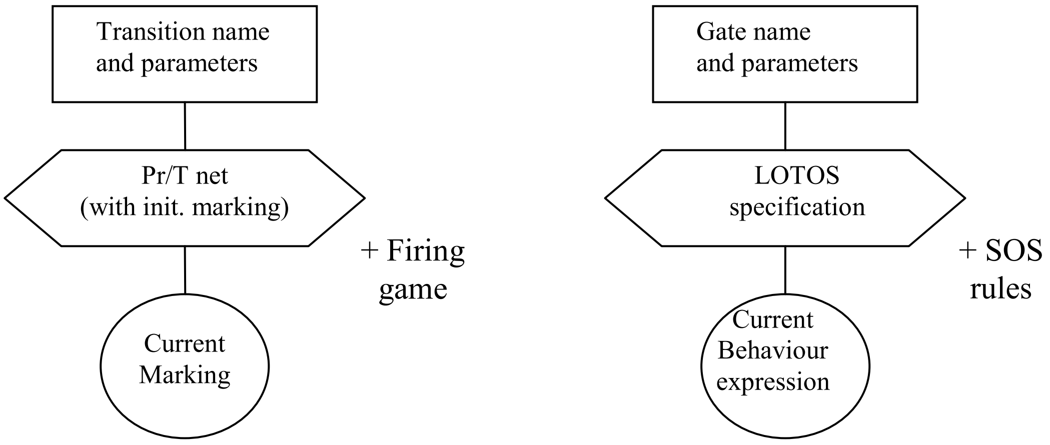
Transition name and parameters
Pr/T net (with init. marking)
+ Firing game
Current Marking
Gate name and parameters
LOTOS specification
+ SOS rules
Current Behaviour expression

Figure 3 - Two instances of the diagram in Figure 1.b: Pr/T nets and Full LOTOS

Note that in both cases the state ranges in an unbounded set of values. A marking may have an unbounded number of tokens, each with values drawn from an unbounded domain. A behaviour expression may grow unboundedly (by using parallel composition in conjunction with process instantiation), and be parameterized by variables of infinite types.

3. Advanced framework

In the previous section a specification was conceived as a single, monolithic piece (see Figure 1.b). We can achieve a better assessment of behavioural specification languages by introducing another dimension in our framework, that has to do with specification decomposition. Formal specifications are produced and read by human beings, who need to break them into manageable blocks. This is where further differences emerge: different specification languages support different system decomposition policies, that are based on different types of specification block, or constraint.

The differentiation among constraint types is basically driven by two distinct factors: the differentiation among event types, or classes, and the fragmentation of the global state. In systems with complex behaviours it is quite natural to distinguish among event types, for accurately modelling functional requirements and user-interaction scenarios. Similarly, one often partitions the global state into pieces, for reflecting logical or physical boundaries within the system.

Taking into account this new dimension leads to the revised framework depicted in Figure 4. Note that the splitting of the event and the global state allows us to explicitly represent also event-to-event and state-to-state constraints.

The purpose of the state-event framework in Figure 4 is to distill and compactly represent the essentials of behavioural description. The meanings of box, circle and hexagon in the diagram are as discussed for Figure 1, except that now the language semantics is left in the background: the hexagon just represents a specification block, or constraint. Let us shortly introduce the five constraint types.

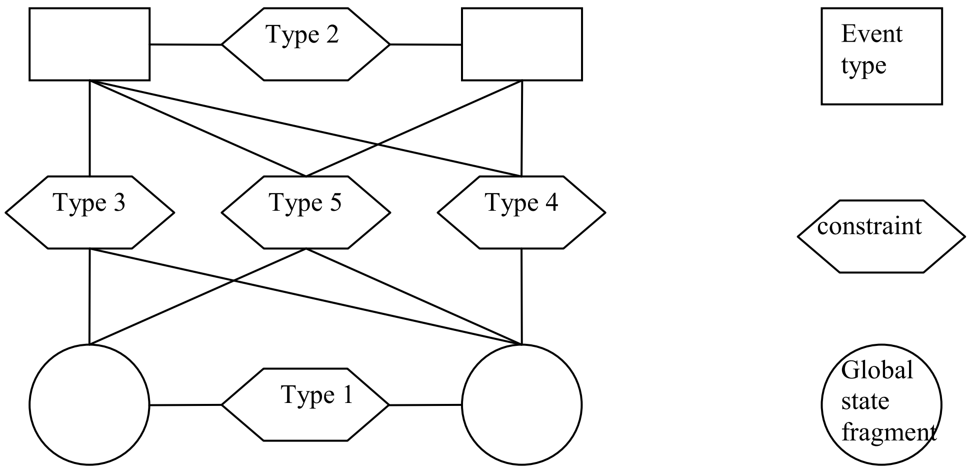

Figure 4 – A conceptual framework: the ST.EVE square

1. **Invariants.**
   The hexagon labelled 'Type 1' connects two state variables (could be, in general, any number), and is meant to represent specification chunks that constrain the possible values simultaneously taken by those variables during system operation. These state-to-state constraints are widely known as state invariants.

2. **Pure event constraints**.
   The hexagon labelled 'Type 2' connects two event types (could be, in general, any number), and is meant to represent specification chunks that constrain the possible instances (occurences) of those events. Recall that each event type could be instantiated several or infinite times, thus yielding multiple event *occurrences*. The purpose of the constraints is exactly to express the 'when' (in terms of relative ordering) and possibly the 'what' (in terms of data values) of event instances. Note that the 'what' is here limited to the consideration of event values, if any, not state values. Thus, event-event relations may be used for expressing aspects such as temporal ordering (sequence), causality, cyclic behaviour, choice, synchronisation, simultaneity, independence, interleaving, priority. Furthermore, we certainly allow for hierarchical definitions, although this is not made explicit in the graphics: complex constraints can be defined in terms of simpler ones.

3. **Disjoint-events/shared-variables.**
   The hexagon labelled 'Type 3' connects exactly one event type and two (could be more) global state fragments, and is meant to represent specification chunks that constrain the

occurrence of events of that type by relating them, via *pre-conditions* and *post-conditions*, to the connected portion of the global state. Note that pre- and post-conditions refer exclusively to state variables, and possibly to the parameters of the event under consideration, not to other events. In other words, information about the history of past events is exclusively available via the current system state. Figure 5.a represents the idea that, in this scenario, constraints are in one-to-one relation with event types, but may share state variables. One could say that the constraints interact by a *shared-variable* policy.

4. **Shared-events/disjoint-variables.**
   The hexagon labelled 'Type 4' connects two event types (could be more) and exactly one portion of the global state, intended as disjoint from those connected to other constraints in the specification. Similar to the pure event-event constraint, this type of specification chunk constrains the possible instances of several event types. However, each constraint can now encapsulate a disjoint portion of the global state, and use it, in conjunction with the connected events, for expressing pre- and post-conditions on event occurrences. The constraints interact purely by a *shared-event* policy (hand-shake, or rendez-vous). This circumstance is pictorially represented in Figure 5.b.

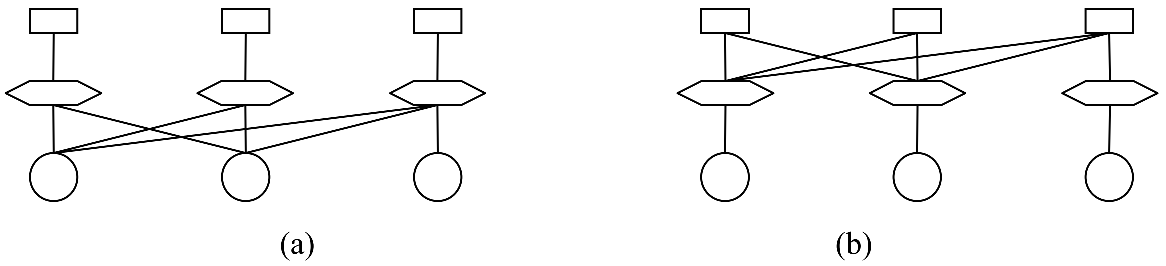

Figure 5 – (a) disjoint-events/shared-variables, and (b) shared-events/disjoint-variables

Figure 6 is a simple transformation of Figure 5 in which the graphical elements in one-to-one relation are collapsed: constraints collapse into events in case (a), and state variables collapse into constraints in case (b). These diagrams are more easily recognized as representative of various specification paradigms, as will become clear later when we consider concrete examples.

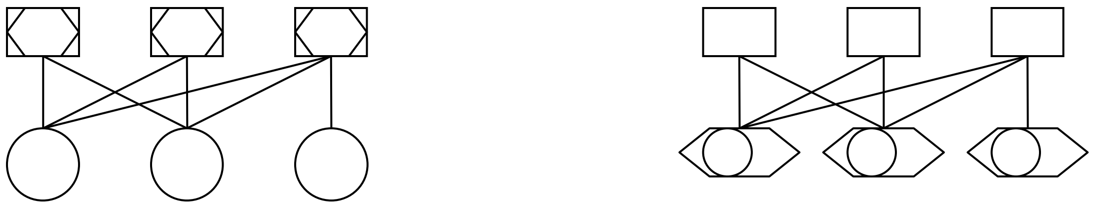

Figure 6 – (a) disjoint-events/shared-variables, and (b) shared-events/disjoint-variables

5. **General case.** The hexagon labelled 'Type 5' is meant to represent constraints that can freely insist on any portion of the global state and any subset of event types, and share both events and state variables. This case represents the least restricted type of specification chunk.

4. Simple constraints

In this section we discuss in more detail the elementary constraints of types 1 and 2 identified in Figure 4, and we mention how they are actually instantiated by some formal languages.

4.1 Invariants

We use this type of constraint for expressing relations that must be preserved among state variables regardless of the events that may change them. Figure 7 illustrates a set of four state variables and a set of four invariants that relate them pairwise. *LivesIn* and *worksIn* are relations in *People* × *Cities*. Whatever event may occur that changes the values of one or more variables, these must keep satisfying the constraints for the whole duration of a possibly infinite system run. According to the specification, not all persons must necessarily work, and not all populated cities must necessarily have people that work there. *Dom* and *ran* denote, respectively, the domain and range of a binary relation.

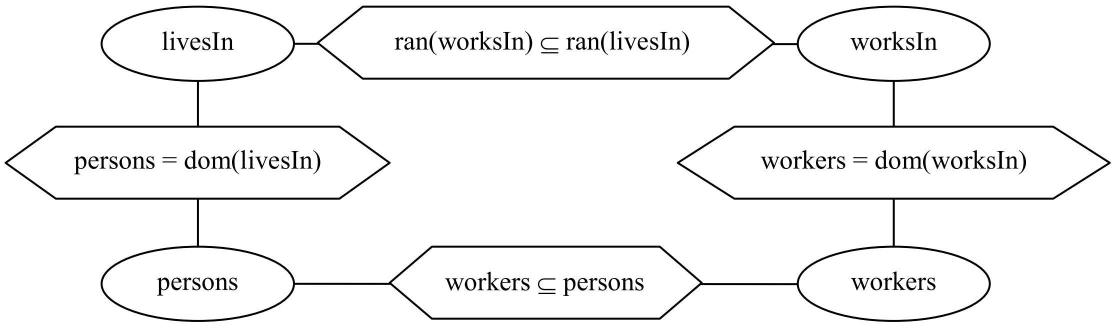

Figure 7 – Four state invariants

Several formal languages support the explicit formulation of invariants: examples are Z, B, and TLA.

Remarks

No realistic specification (language) can limit itself to the pure expression of invariants. The identification of the state variables and the relations that must be preserved as the system evolves may be an convenient starting point; however, this provides no information about what can actually happen, that is, about the nature and temporal sequence of events. For this reason, some specification techniques, B for example, adopt invariants as a sort of initial contract that the subsequent, more detailed behavioural specification is expected not to violate.

4.2 Pure event constraints

What is an event? Probably the most simple and general definition is suggested by physics: an event is a point in space and time. The pair (room B, 9.00 am) represents an event, one that happens in room B at 9.00 am. Assuming a one-dimensional space, an event is a pair (x, t), where *x* and *t* real numbers. Figure 8 illustrates a set of four events and a set of four constraints that relate them pairwise. Events (x1, t1) and (x2, t2) occur on a train running at constant speed, respectively at the front and rear of it: their space distance is *TrainLength*. Events (x1', t1') and (x2', t2') occur on the ground, simultaneously: their time distance, relative to the ground system, is zero. If we insist that (x1, t1) and (x1', t1') are the same event as seen, respectively, from the train and from the ground, their coordinates must be related by the Lorentz transformation $\mathrm{Lorentz}_V$(x, t, x', t'):

$$x = \frac{x' - v.t'}{\sqrt{1 - v^2 / c^2}} \qquad t = \frac{t' - (v / c^2) x'}{\sqrt{1 - v^2 / c^2}}$$

where *v* is the speed of the train and *c* is the speed of light. The same applies to events (x2, t2) and (x2', t2').

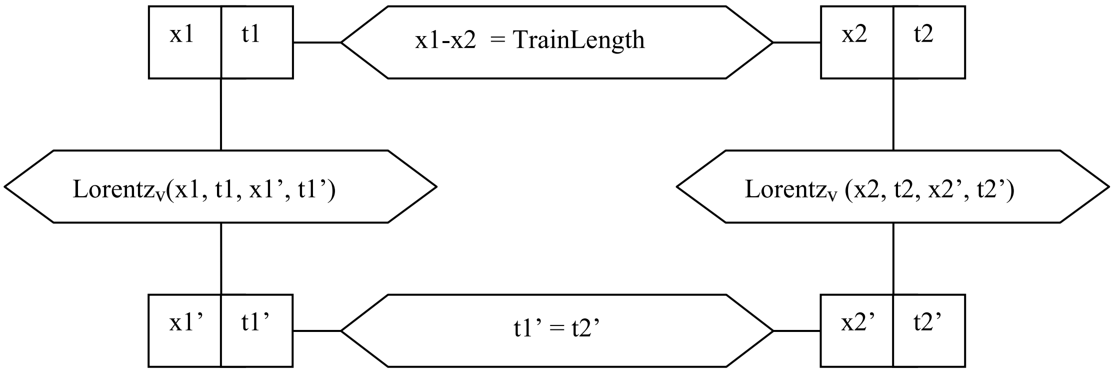

Figure 8 – Events and constraints for measuring the length of a running train

.
The diagram can indeed be seen as a behavioural specification: the system behaviour is composed by any quadruple of events that satisfies the constraints. These are the essential events involved in the experiment of measuring the length of a running train, as described in [E16]. In particular, if the constraints are respected, the space distance between events (x1', t1') and (x2', t2') always provides the length of the train as measured from the ground, which turns out to be shorter than the *TrainLength* measured from the train.

The limit of this specific instantiation of pure event constraints is that the solution of the network of constraints only accounts for finite sets of actual events, since the boxes actually represent event *instances*. Unlike the case of state invariants, every alternative solution here is *another* finite system run, and infinite behaviours are out of reach.

For this reason, many formal specification languages omit time information from events, or just use relative rather than absolute time measures, and succeed in supporting finite representations of infinite event occurrences. Examples are CCS, CSP and (Basic) LOTOS. For example, the four constraints in Figure 9 can be expressed by the four LOTOS processes defined below. Each constraint/process captures a fragment of information about the relative orderings of two parameter-less events. Although in principle one could have breakfast on the way to the office, due to the need to feed the dog at home, but only after breakfast, the latter is taken at home.

```
Process P1[wakeUp, breakfast]    := wakeUp; breakfast; P1[wakeUp, breakfast]
Process P2[wakeUp, drive]        := wakeUp; drive; P2[wakeUp, drive]
Process P3[breakfast, feedDog]   := breakfast; feedDog;P3[breakfast, feedDog]
Process P4[feedDog, drive]       := feedDog; drive; P4[feedDog, drive]
```

The processes can then be composed by the LOTOS parallel expression:

```
(P1[wakeUp, breakfast] |[wakeUp]| P2[wakeUp, drive])
|[drive, breakfast]|
(P3[breakfast, feedDog] |[feedDog]| P4[feedDog, drive])
```

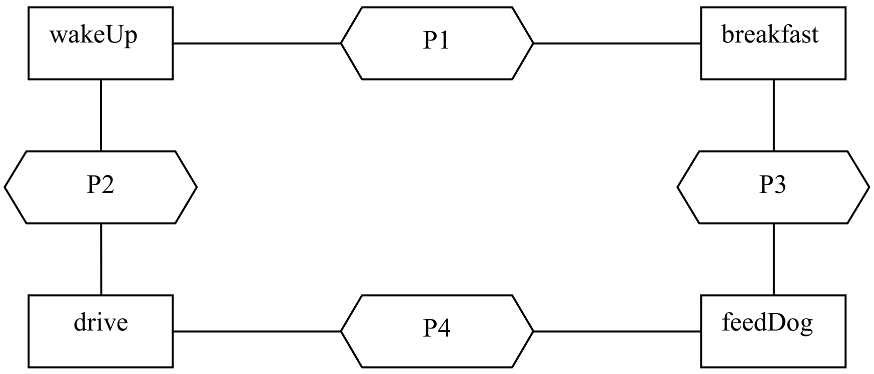

Figure 9 – Pure event constraints for some daily activities

The reader may object that the small specification above is not a valid example of pure event-oriented reasoning, since behaviour expressions represent evolving state information, as discussed Section 2. However, we insist in ascribing *pure* behaviour expressions to pure event-oriented thinking. A *pure* behaviour expression is one that does not manipulate data. In fact, in Basic LOTOS one can't write anything but pure behaviour expressions. And indeed, while the example above makes use of only a few behavioural operators (namely action prefix, process instantiation and parallel composition), the set of expressive tools for supporting pure event-oriented specification is wider, and includes operators such as choice, interleaving, enabling, disabling.

We claim that pure event-oriented specification can indeed be achieved also by making explicit use state variables, as long as (i) these assume finitely many values, and are used only for event-ordering purposes, not for representing data structures, and (ii) the behaviour is described in terms of instances of a finite set of parameter-less events (as in the example above). For example, one can readily replace the four Basic LOTOS processes above by equivalent, two-state machines. And, in the same way in which one can compose Basic LOTOS processes in a number of ways, via choice, interleaving, partial synchronisation, enabling, disabling, and so on, one can consider equivalent, or similar operations on state machines, along the lines of Statecharts [H87].

Remarks

Is pure event-oriented specification any useful? Several researchers and developers tend to answer negatively, based on the idea that a good model for the global system state is a necessary step for building a useful specification. On the contrary, our experience with applications of process algebras, in their data-less forms, seems to indicate that pure event-oriented thinking does offer some advantages, although these only apply to the *very* early stages of system conception .

Let us first point out the typical limitation of this method: by reasoning solely in terms of temporal ordering of pure events, we cannot record any data and refer to it later for deciding about the possible occurrence or value of some event. Consider the evergreen example of a vending machine. At a very high abstraction level, its behaviour can be described in terms of a small number of pure events, such as *InsertCoin, SelectItem, GetChange, GetChocholate, GetTea*, and the like. In a pure event-oriented description, we would not be allowed to refine, say, the *InsertCoin* events by numeric parameters indicating coin values, nor could we accumulate these values in an internal variable, for later deciding whether or enough money was inserted for enabling a *GetChocolate* event. Actually this limitation could be circumvented by coding (finitely many) values into event names, yielding names such as *Insert5, Insert10, Insert20, Insert50*, … and by growing event trees in a way which carefully keeps track of the number of different *Insert* events up to any stage along each path. But, due to combinatorial explosion, very soon this approach would lead to unmanageable complexity. On the contrary, a combined usage of parameterized events, such as *Insert(x: Coin)*, and state variables, such as *CurrentDeposit*, would yield a simple description. A state variable is enough for keeping track of the relevant events, abstracting from the order in which they have occurred.

In conclusion, we believe that pure event-oriented thinking can be used mainly for providing *very* abstract, and *very* early behavioural descriptions of a wide class of reactive systems. They would be even more useful if formal refinement techniques were available for smoothly introducing state information. Often the supporters of state-oriented thinking criticize process algebraic approaches by saying that they cannot scale up to realistic system sizes, and that they are only good at describing toy examples such as vending machines. Our reply is that a vending machine is indeed a very complex object, as demonstrated by its frequent malfunctioning; it could be described by very complex internal state structures and event patterns. It is the specifier's choice of a very high abstraction level that makes the *description* so simple. Whether the level is too high, and the description too naïve to be of any practical use, even for pure documentation purposes, is another question, which is left open for further discussion.

5. Complex constraints

In this section we discuss in more detail the constraints of types 3 and 4 identified in Figure 4, and we mention how they are actually instantiated by some formal languages.

5.1 Disjoint-events/shared-variables

The structure of specifications of this type is abstractly depicted in Figures 5.a and 6.a. These specifications are structured around event types: the specification is a set of formal fragments, or blocks, each describing one event type and its associated pre- and post-conditions, which involve a number of state variables. Events are usually expected to occur one at a time. Specification blocks may share state variables.

Many behavioural formal models adopt this specification paradigm, possibly combined with invariants. Z, B, TLA are fundamentally based on pre- and post- conditions and invariants. For example, in Z one can write a schema describing the global state and its invariants, and other schemata, one for each operation. These identify pre- and post-conditions on state variables, and input and output parameters associated with the operation.

Predicate/Transition nets express pre- and post-conditions on events in a very appealing, partly graphical way. The net of Figure 10, that matches Figure 6.a, describes a portion of a car share system. Places *livesIn* and *worksIn* represent relations in *People* × *Cities* (as in Figure 7): each token in these places is an element of the relation, that is, a (person, city) pair. The values of the tokens requested and produced by transition *'Pairing'* are indicated in the multisets of terms (pairs) appearing in the shown arc inscriptions. The transition is characterized also by an input parameter – the system user indicates the city *c* where the involved persons *p1* and *p2* must live -- and includes a predicate ($c \neq d$) establishing that the city where they work is different, so that it makes sense for them to share a car.

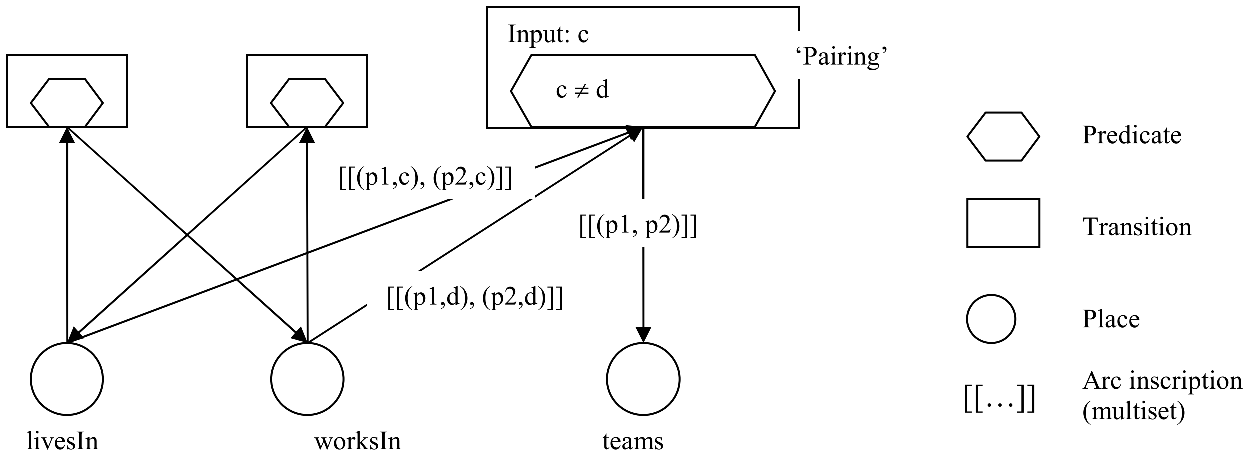

Figure 10 – A detail of a predicate/transition net for a car share system

Remarks

The success of the disjoint-event/shared-variables approach, using pre- and post-conditions, is due to its simplicity: it allows one to concentrate on event types individually. One does not have to think explicitly about patterns of events in time: these are implicitly defined by the 'game' of pre-conditions and post-conditions. Specifiers only define the rules of this game. The advantage of this approach is also its limitation: one cannot directly express 'views' about system behaviour made up of several events.

5.2 Shared-events/disjoint-variables

The structure of specifications of this type is abstractly depicted in Figures 5.b and 6.b; they are structured into chunks that describe the behaviour of one logical or physical entity, each encapsulating a portion of the global state and insisting on a number of events. Each fragment – possibly called a *process* -- expresses ordering and other constraints on the occurrences of these events. Entities interact by rendez-vous (shared events).

This is the specification paradigm adopted by process algebraic languages such as CCS and CSP. The temporal constraints on events are expressed by behaviour expressions, built in terms of behavioural operators. Mutual influences between event parameters and data values are also expressible, whenever these languages offer data representation capabilities.

Figure 11, matching Figure 6.b, illustrates three LOTOS processes *People*, *Cities* and a non-specified *R*, that interact by sharing events *birth*, *death*, and a non-specified event *g*. Events *birth* and *death* have the same structure: they have two parameters, namely a person and a city. The LOTOS syntax for the composition is:

```
(People[birth, death, g](P)
|[birth, death, g]|
Cities[birth, death, g](C)
)
|[g]|
R[g](X, Y)
```

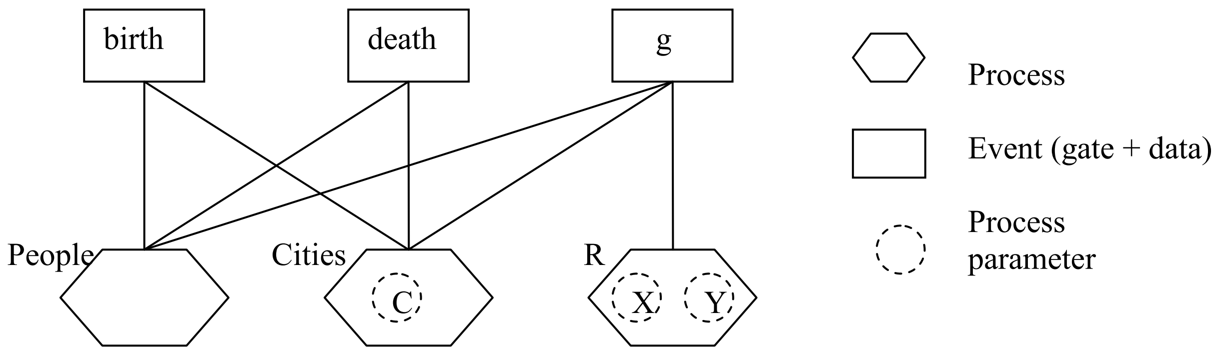

Figure 11 - Interacting LOTOS processes

Minimal definitions for processes *People* and *Cities* could be as follows. We use underscore '_' for don't care variables.

```
Process People[birth, death] :=
   OnePerson[birth, death]
   |||
   i;  People[birth, death]

   where
   Process OnePerson[birth, death] :=
         birth ?p: person ?_: city;
         death !p ?_: city;
         stop

Process Cities[birth, death](C: setOf City) :=
      birth ?_: person ?aCity: city;
         ( [aCity In C] ->      Cities[birth, death](C)
         [][aCity NotIn C] ->   Cities[birth, death](C union [aCity])
         )
   [] death ?_: person ?aCity: city;
         Cities[birth, death](C minus [aCity])
```

Process *Cities* handles the set *C* of cities currently represented in the system. LOTOS variables are not exactly state variables, as found in imperative languages: they do not represent memory locations that can be re-assigned values. Each process is parameterized by 'logical' variables, che get bound to the values passed via actual parameters when the process is instantiated; updating these variables is achieved by re-instantiating the process with different actual parameters. These variables are represented in Figure 11 by dotted lines.

An object-oriented instance of the paradigm of communicating agents that share events and encapsulate state information, where the latter is represented as traditional, imperative state variables, is described in [BD98].

6. Design choices behind a new mixed specification model

In this section we consider the recently introduced model of *Abstract State Processes* [BB03], and match its features against the conceptual framework introduced so far, with the purpose to better assess the design choices behind its definition. Abstract State Processes enrich ASM (Abstract State Machines) with few behavioural operators borrowed from process algebra, for supporting flexible specification of concurrent, distributed, reactive systems.

As a first step, in Figure 12 we instantiate the general diagram of Figure 1.b w.r.t. Abstract State Processes.

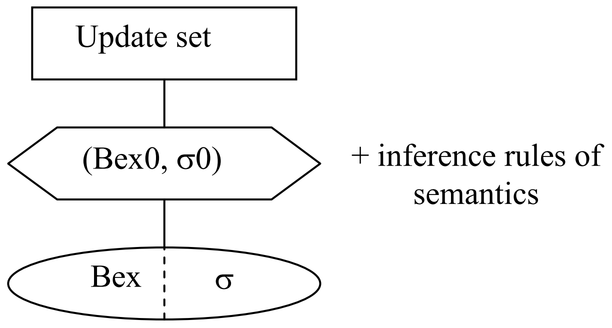

Figure 12 – Instantiating the diagram of Figure 1.b for Abstract State Processes

The diagram reflects the following design choices.

- **Event**. In Abstract State Processes, an *event* is an update set, that is, a set of pairs, each consisting of a memory location and a value to be assigned to it. The updates take place synchronously, and affect the current state σ. The notion of synchronous updates is exactly as found in the original ASM model.
- **State**. In Abstract State Processes, a state is a pair (B, σ), called *configuration*, where B is a behaviour expression B and *σ* is a traditional ASM state, that is, a structure of functions (we shall still use the term 'state' for referring to this element). Thus, in Abstract State Processes we combine two fundamental ways to represent state information, as adopted, respectively, by imperative languages and by process algebra (both were represented in Figure 3)

The inference rules of the abstract state process semantics [BB03] allow one to derive transitions of the general form:

(B, σ) --- u ---> (B', σ')

where (B, σ) is the current system configuration, *u* is an update set (an event), and (B', σ') is the next configuration. The state *σ'* is obtained from *σ* by applying the update set *u* to *σ'*, exactly as done for ASMs.

We consider now the problem of relating Abstract State Processes with the conceptual framework of Figure 4. In this formalism, a system is conceived as a collection of interacting processes, and a specification is structured as a collection of process definitions, representing the constraints of our framework. In the current definition of Abstract State Processes [BB03], any process in a specification can refer to any element of the *σ* state, and update it. Since events *are* update sets, a process sharing with other processes portions of the state will share, in general, also events, unless read-only variables are explicitly introduced. Let us be more precise on the way in which two abstract state processes may share an event. Similar to ASM, two parallel processes *P1* and *P2* share an event/update set *U*, when *P1* can produce event *U1*, *P2* can produce event *U2*, and *U* = *U1*

∪ *U2*. *U1* and *U2* must update in the same way the locations they share (if any). Furthermore, the *selective synchrony* parallel composition of Abstract State Processes, borrowed from CSP and LOTOS, implies synchronisation, and the union of update sets, in a more selective way. Only when the composed processes are ready for events that include the updating of a common location explicitly indicated by the user in the operator itself, do the two processes synchronize; otherwise, their events are interleaved. The composition 'P |[f]| Q' means that *P* and *Q* may produce a joint event only if are both ready to consistently update, possibly among other locations, the same point of function *f*. For example, *P* may be ready to produce the updates *g(x) := a* || *f(y) := b,* and *Q* may be ready to produce the updates *f(y) := b*|| *h(z) := c*; then, their composition yields the updates:

g(x) := a || f(y) := b || h(z) := c

Based on the above discussion, and referring to Figure 4, we would be induced to view Abstract State Processes as type 5 constraints, and to represent them as in Figure 13.

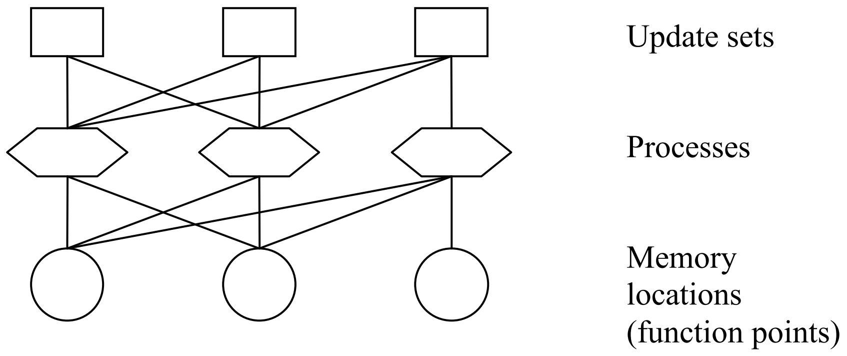

Figure 13 – Interacting abstract state processes

However, while diagrams such as the one in Figure 13 suggest (as precisely intended) that events and state components are equally important factors in structuring a specification, Abstract State Processes do not fully reflect this symmetry. When understood as update sets, events are conceptually subordinate to the state component, and it does not make sense to think of defining and partitioning the event space of a system *before* providing its state structure. And, even when the state structure is fully defined, the different combinations of event types (different tuples of locations to be simultaneously updated) that may emerge at execution time tend, at least in principle, to explode combinatorially: event types do not lend themselves as a natural means for structuring the specification.

In conclusion, Abstract State Processes do not seem to find a fully convincing place in the state-event framework we have successfully used for the other formalisms mentioned in this paper. The reason is that the notion of event with which we have equipped them is not sufficiently independent from that of state. Do we then need a new, stronger notion of event for Abstract State Processes?

7. Conclusions

We have introduced, in two steps, a conceptual framework for assessing formal specification languages with respect to their ability to model complex system behaviours. The framework explicitly handles the primitive concepts of state, event, and five types of constraint. Event-event, state-state, and state-event constraints correspond to the various constructs offered by specification languages for decomposing system specifications into manageable chunks (schemata, rules, processes, blocks…). We believe that recognizing explicitly these fundamental and different ways of thinking about system behaviours *before* selecting a formal specification model should help in making more conscious decisions in the upper phases of software development. Our experience indicates that often software engineers and computer scientists express their opinions and preferences on modeling techniques without referring explicitly to the simple ideas and options that we have tried to distill into the 'st.eve' framework.

Several questions are posed by this paper, and left open. Is the most general form of constraint – type 5 of Figure 4 – any useful? In other words, are there advantages, in terms of expressive flexibility, in working with specification units that may share simultaneously state variables and events? We suspect that the increased expressive potential offered by this mix might not be of much practical use, since specifiers would miss the divide-and-conquer strategies based on event types or, separately, on state encapsulation, and would be left with no clear guidance for specification decomposition. For this reason, many methods only admit constructions of type 3 or type 4, that are proper sub-cases of type 5: the apparent limitation is indeed adopted for promoting discipline in writing specifications (this reminds us of a saying attributed to the French composer Pierre Boulez: 'When everything is allowed, nothing is possible').

In the very early phases of system conception, it may happen that constraints of mixed types are incrementally produced. For example, after a very abstract, stateless specification given in terms of pure events, one may provide a refined specification involving the state structure and a refinement of the events with data parameters. Much remains to be investigated about the practical usefulness of these mixed specifications, and about the associated consistency and verification problems.

On a longer run, an attractive problem is the relation between natural language descriptions of system requirements and the different formal specification paradigms. Appropriate restrictions of natural language are likely to facilitate one or the other type of formalisation.

References

[AL93] M. Abadi, L. Lamport, 'Composing Specifications', *ACM Transactions on Programming Languages and Systems* 15, 1 (January 1993), 73-132.

[A96] J.-R. Abrial, The B-Book – Assigning Programs to Meanings, Cambridge University Press, 1996.

[BB87] T. Bolognesi, E. Brinksma, 'Introduction to the ISO Specification Language LOTOS', *Computer Networks and ISDN Systems*, Vol. 14, No. 1, pp. 25-59, *North-Holland*, 1987.

[BB03] T. Bolognesi, E. Boerger, 'Abstract State Processes', in: E. Boerger, A. Gargantini, E. Riccobene (eds), *Abstract State Machines - Advances in Theory and Applications, Proceedings of 10th International Workshop, ASM 2003, Taormina, Italy, March 2003,* LNCS 2589, Springer-Verlag 2003..

[BD98] T. Bolognesi, J. Derrick, 'Constraint-oriented style for object-oriented formal specification'*, IEE Proc.-Soft.*, Vol. 145, No. 2-3, April-June 1998.

[BRJ99] G. Booch, J. Rumbaugh, I. Jacobson, The Unified Modeling Language User Guide, Addison-Wesley, 1991.

[BS03] E. Boerger, R. Stark, Abstract State Machines, Springer-Verlag 2003.

[E16] A. Einstain, Űber die spezielle und allgemeine Relativitätstheorie (gemeinverstandlich), 1916.

[G93] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Boerger, editor, Specification and Validation Methods, pages 9–36. Oxford University Press, 1995.

[H85] C. A. R. Hoare, Communicating Sequential Processes, Prentice-Hall, 1985.

[H87] D. Harel, 'Statecharts, a visual formalism for complex systems', *Science of Computer Programming,* 8, North-Holland, 1987.

[JR91] K. Jensen, G. Rozenberg, High-Level Petri Nets, Springer-Verlag, 1991.

[L03] L. Lamport, Specifying Systems, Addison-Wesley, 2003.

[M80] R. Milner, A Calculus of Communicating Systems, Lecture Notes in Computer Science, Vol.92, Springer-Verlag, 1980.

[S89] J. M. Spivey, The Z Notation – A Reference Manual, Prentice-Hall, 1989.