Under consideration for publication in Formal Aspects of Computing

A Semantics of Simultaneous Events (SSE) for State-Based Formal Specification[1]

Chris Taylor[1] and David Till[2]

[1]Computing Laboratory, University of Kent at Canterbury, Canterbury, Kent, CT2 7NF, UK
[2]Department of Computing, City University, Northampton Square, London, EC1V 0HB, UK

Keywords: State-based formal specification; Non-interleaving concurrency; The frame problem

Abstract. Simple state-based models use events corresponding to sets of prestate, poststate pairs of system states, and solve the frame problem — i.e. knowing what else has changed, given that an event has occurred — by assuming non-concurrent events that uniquely determine the poststate. However, such events are "overspecified" and thus not reusable in other models. This paper describes a more flexible approach to state-based concurrency and the frame problem, in which events — loosely specified in semantic terms — may occur simultaneously and can be reused in different contexts without "promotion".

1. Introduction

State-based or model-based formal specification — often associated with the languages Z [Spi88, PST96] and VDM [Jon90] — is a well-established paradigm. State-based models describe systems in terms of states and transitions between them, without reference to the timing of state changes or the duration of states. This paper presents a *semantics of simultaneous events* (SSE) for specification of this kind, which allows simultaneous (i.e. non-interleaving) concurrency. Its events are defined independently of a specific context, frame, or invariant, so that they can be reused without "promotion", when systems are composed into larger systems.

It is helpful to relate the semantic models discussed to a very simple example (which is used by Pitt and Byers in [PiB94], and so facilitates comparison with their approach) — a number store which, at any given moment, holds some, all, or none of the three numbers 1, 2 and 3. Formal models of this system should be able to represent events such as "add 1 to the store" and "add 1 to the store, leaving everything else unchanged". The aim will be to allow simultaneous additions and removals from the store, and to capture

[1] This research arose from work funded by the ESPRIT project NATURE on requirements engineering ([NAT97]).
Correspondence and offprint requests to: Chris Taylor, Computing Laboratory, University of Kent at Canterbury, Canterbury, Kent, CT2 7NF, UK (C.N.Taylor-1@ukc.ac.uk).

certain inferences concerning them. For example, from the occurrence of the first event above, it should be possible to infer that 1 has been added, but not — in the absence of other information — whether 2 or 3 has been added or removed simultaneously; whereas from the occurrence of the second event above, it should be possible to infer that neither 2 nor 3 has been added or removed. This kind of problem — of reasoning about what else has or has not changed, given the occurrence of a certain event — is often referred to as the *frame problem*, a term which dates back to the early artificial intelligence literature ([McH69]). Tackling it is a key aspect of any semantics which allows concurrency.

The rest of the paper is organised as follows. Section 2 describes some related semantic models for state-based systems, and discusses the application of two of them to the number store system. Section 3 defines the SSE framework, illustrating it with examples related to the number store. Section 4 examines some further issues arising from SSE, including the parallel composition of systems (using the well-known "dining philosophers" problem as an example), the modelling of events with "inputs" and "outputs", and the relevance to the frame problem of the "granularity" of state components. Section 5 provides a summary and some pointers to future work.

2. Some Related Formal Approaches

This section examines some related formal approaches to state-based specification and the frame problem. The first subsection describes a simple state-transition model, involving deterministic, non-concurrent events — in which the frame problem is trivially solved, but at the cost of inflexibility and "overspecification" of events. The second subsection examines a semantics proposed by Pitt and Byers in [PiB94], which allows successive states to be linked by sets of simultaneous events, rather than by individual events. This shares many of the aims of SSE, but its handling of the frame problem, although sensible in some cases, is counter-intuitive for certain disjunctive events. The third subsection briefly refers to some other related work (particularly in the AI literature), summarising some of the features used by existing specification to help with the frame problem, and discussing the relevance of the frame problem to object-oriented specification languages.

2.1. A Simple Non-Concurrent, State-Transition Model

Conventional state-transition models describe a system by specifying a set of *states*, and a set of *operations* or *events* (types of occurrence) by which successive states can be related. States are often represented as sets of pairs of the form $\{c_1 \mapsto v_1, \ldots, c_n \mapsto v_n\}$, which map typed *state components* $c_1, \ldots, c_n$ to values $v_1, \ldots, v_n$ of the appropriate types. The states of a given system are sometimes restricted by a *state invariant* that mutually constrains the components of those states.

In simple state-transition models, events are just sets of prestate, poststate pairs, which are assumed to be both *mutually exclusive* (i.e. the occurrence of one event precludes the simultaneous occurrence of any others) and *deterministic* (i.e. if a given event occurs in a given state, there is only one possible poststate). In one variant of this approach, a system *sys* can be defined as a tuple $(S_{sys}, I_{sys}, E_{sys})$, where:

- S_{sys} is a finite set of *system states* ($\varnothing \subset S_{sys}$).
- I_{sys} is a finite set of *initial states* ($\varnothing \subset I_{sys} \subseteq S_{sys}$).
- E_{sys} is a finite set of events ($\varnothing \subset E_{sys} \subseteq (S_{sys} \times S_{sys})$), such that for any distinct events e_1 and e_2, $e_1 \cap e_2 = \varnothing$ (mutual exclusion), and for any event e and any states s, s', and s'', if $(s, s') \in e$ and $(s, s'') \in e$, then $s' = s''$ (deterministic events).

The behaviour of such a system can be characterised by a set of infinite sequences of states

$$s_1 \; s_2 \; s_3 \ldots$$

where s_1 is a permitted initial state, and the nth state s_n and its successor s_{n+1} are related by some event e (i.e. $(s_n, s_{n+1}) \in e$); or by the corresponding labelled transition sequences

$$s_1 \xrightarrow{e_1} s_2 \xrightarrow{e_2} s_3 \ldots$$

which show the linking events. (Relative to a given set of mutually exclusive events, the labels are redundant.) In more complex variants of such systems, extra constraints may be imposed, e.g. that every state should satisfy a certain invariant, that every state should be reachable in some sequence, or that the occurrences of events should satisfy fairness constraints.

Although individual events are deterministic in such a system, the overall behaviour may not be, because in some states, several alternative next events may be possible. If E_{sys} includes the identity relation on the set S_{sys}, then "stuttering" is possible, i.e. the same state can persist for several consecutive steps.

In this simple approach, the frame problem is trivially solved, because events cannot occur simultaneously, and the occurrence of a given event in a given prestate completely determines the poststate. However, the cost of handling the frame problem in this way is a rather inflexible and limited semantics.

Consider the simple number store of section 1. Pitt and Byers [PiB94] choose to define its possible states as the subsets of $\{1, 2, 3\}$, so that e.g. $\{1, 2\}$ means that 1 and 2 are stored but 3 is not. This amounts to using a state $\{in \mapsto \{1, 2\}\}$ with a single component in, but since a solitary component is rather redundant, one may just as well use its value alone to represent the state. There are, of course, other ways of modelling the states of the number store — for example, section 3 develops a model that uses three Boolean-valued state components $in1$, $in2$, and $in3$. The implications of such arbitrary modelling choices as regards the frame problem are examined in detail in section 4.3.

If states are represented as subsets of $\{1, 2, 3\}$, and events are required to be deterministic, then individual additions and removals from the store can only be represented by events such as:

$$just\_add1 = \{ (s, s') \in S_{sys} \times S_{sys} \mid 1 \notin s \wedge 1 \in s' \wedge s \cap \{2,3\} = s' \cap \{2,3\} \}$$

This event states that 1 has been added, but also that the membership status of 2 and 3 has not changed — for if it did not make explicit what happens to 2 and 3, there would be more than one poststate from a given prestate, and the event would not be deterministic. Similarly, the simultaneous addition of 1 and 2 can be represented only as a single event (which leaves the status of 3 unchanged):

$$just\_add1\&add2 = \{ (s, s') \in S_{sys} \times S_{sys} \mid s \cap \{1,2\} = \varnothing \wedge \{1,2\} \subseteq s' \wedge s \cap \{3\} = s' \cap \{3\} \}$$

So, using deterministic, mutually exclusive events solves the frame problem, but forces events to be "overspecified", and prevents the representation of concurrency in terms of distinct, simultaneous events. Now suppose the requirements on events were relaxed, allowing more loosely specified events such as:

$$add1 = \{ (s, s') \in S_{sys} \times S_{sys} \mid 1 \notin s \wedge 1 \in s' \} \qquad add2 = \{ (s, s') \in S_{sys} \times S_{sys} \mid 2 \notin s \wedge 2 \in s' \}$$

These events can occur simultaneously, since some state pairs — e.g. $(\{3\}, \{1, 2, 3\})$ — are related by both. They are also non-deterministic — e.g. both $(\{3\}, \{1, 3\})$ and $(\{3\}, \{1\})$ are related by $add1$, since $add1$ says only that 1 becomes a member, but says nothing about what happens to 2 or 3. Such loosely specified events offer greater flexibility, but any semantics which allows them faces the frame problem in a non-trivial form, and must provide some means of constraining the poststates of individual events and sets of simultaneous events.

2.2. The Pitt–Byers Semantics for Concurrent State-Based Systems

In [PiB94], Pitt and Byers propose a semantics for state-based systems (based on earlier work by Schuman, Pitt and Byers in [ScP86] and [SPB90]), which allows loosely specified, simultaneous events like those just described, and provides a mechanism for handling the frame problem. They define a *relational/neutral system* as a tuple (W, E, N, C), consisting of:

- A set W of *system states* ($\varnothing \subset W$).
- A set E of binary relations on W called *events* ($\varnothing \subset E \subseteq \mathbb{P}(W \times W)$).
- A set N of binary equivalence relations on W called *neutral events*, closed under intersection and union of its members, and application of C ($\varnothing \subset N \subseteq \mathbb{P}(W \times W)$).
- A "complement" function $C : N \to N$.

E and N may intersect. N must form a Boolean algebra under intersection, union, and application of C, with $W \times W$ and the identity relation I on W as identity elements for intersection and union respectively. For any neutral n, the intersection of n and $C(n)$ is I, and the union of n and $C(n)$ is $W \times W$.

The simple number store of section 1, given by Pitt and Byers, can be modelled as a relational/neutral system by defining W as the set of all subsets of $\{1, 2, 3\}$ and E as the set of all binary relations on W (i.e.

$E = (W \times W)$). In each state s belonging to W, each of the numbers 1, 2 and 3 is either a member of s, or not. N is defined as the set of equivalence relations n_A on W that leave unchanged the membership or non-membership of some particular subset A of the three numbers. Formally, n_A, N and the complement function C are defined by:

$$n_A = \{ (s, s') \in W \times W \mid s \cap A = s' \cap A \}$$
$$N = \{ n_A \mid A \subseteq W \} \qquad\qquad C(n_A) = n_{W \setminus A}$$

For example, states s and s' are related by the neutral $n_{\{1\}}$ iff either 1 belongs to both s and s', or 1 belongs to neither. The state pairs $(\{1,2\},\{1\})$ and $(\{2\},\{2,3\})$ are amongst those that satisfy this condition, whereas $(\{1,2\},\{2\})$ and $(\{2\},\{1,2\})$ are not. Thus $n_{\{1\}}$ can be described as "leave the membership status of 1 unchanged". The complement $C(n)$ of a neutral n is that neutral which allows to change what n requires not to change, and vice versa — e.g. $C(n\{1\})$ is $n_{\{2,3\}}$, i.e. "leave the membership status of 2 and 3 unchanged".

Neutrals are used to reason about what "stays the same" if events belonging to E occur, and thus provide a mechanism for handling the frame problem. For any neutral event n and event e (neutral or otherwise), n is *consistent* with e iff in any prestate in which e can occur, n can occur as well, without ruling out the possibility of a poststate; and n is *central* to e iff consistent with $e \cap n'$ for all neutrals n' consistent with e. The neutrals central to e are mutually compatible neutrals consistent with e.

If an event e belonging to E occurs whilst "the rest stays unchanged", then the poststates possible from a given prestate s are defined as all those related to s both by e, and by all neutrals central to e.

Although Pitt and Byers do not use such a syntax, their analysis could be expressed in terms of a *just* operator roughly analogous to SSE's $just_W$ and $just_S$ operators, defined in section 3.5. (SSE's two operators differ slightly in their relation to state invariants: no such distinction is needed here, because a system (W, E, N, C) does not specify a state invariant.) Relative to a particular relational/neutral system, this could be defined as follows, for any event e:

$$just(e) = \{ (s, s') \in e \mid \forall n \in central(e) \bullet (s, s') \in n \}$$

Here, $just(e)$ — that event which occurs when "e occurs, and the rest stays unchanged" — consists of all prestate, poststate pairs related by e, which are also related by all neutral events "central" to e. The operator can also be applied to the simultaneous conjunction of a set E' of events, by using set intersection to represent conjunction, i.e. by writing $just(\bigcap E')$.

The state sequences of a relational/neutral system (W, E, N, C) are not explicitly defined in [PiB94], but it is clear from [SPB90] that a process called *historical inference* is used to specify them. Such a sequence must begin with a permitted initial state, and thereafter, each state must be related to its successor by $just(\bigcap E')$, for some subset E' of E. In other words, each state transition is labelled with a set of simultaneous events, rather than with a single event, and the neutrals central to the conjunction of those events are used to restrict what else has changed.

For example, suppose the events $add1$ and $add2$ are defined as in section 2.1 (except that Pitt and Byers use W, not S_{sys}, for the set of system states). Then $consistent(add1)$ and $central(add1)$, denoting respectively the sets of all neutrals consistent with and central to $add1$, are as follows:

$$add1 = \{ (s, s') \in W \times W \mid 1 \notin s \wedge 1 \in s' \} \qquad add2 = \{ (s, s') \in W \times W \mid 2 \notin s \wedge 2 \in s' \}$$
$$consistent(add1) = central(add1) = \{n_\varnothing, n_{\{2\}}, n_{\{3\}}, n_{\{2,3\}}\}$$

Here, the four consistent neutrals are all central, and fix the membership status of respectively, nothing, 2, 3, and both 2 and 3. The last of these neutrals implies the others, because fixing the membership status of both 2 and 3 implies fixing their statuses individually — and $n_\varnothing$ imposes no constraint at all, since $n_\varnothing = W \times W$. Thus:

$$\begin{aligned}
&just(add1)\\
&= \{ (s, s') \in add1 \mid \forall n \in central(add1) \bullet (s, s') \in n \}\\
&= \{ (s, s') \mid 1 \notin s \wedge 1 \in s' \wedge \forall n \in \{n_\varnothing, n_{\{2\}}, n_{\{3\}}, n_{\{2,3\}}\} \bullet (s, s') \in n \}\\
&= \{ (s, s') \mid 1 \notin s \wedge 1 \in s' \wedge (s, s') \in n_{\{2,3\}} \}\\
&= \{ (s, s') \mid 1 \notin s \wedge 1 \in s' \wedge s \cap \{2,3\} = s' \cap \{2,3\} \}\\
&= \{ (\varnothing, \{1\}), (\{2\}, \{1,2\}), (\{3\}, \{1,3\}), (\{2,3\}, \{1,2,3\}) \}
\end{aligned}$$

The resulting event here is intuitively sensible, because it says that nothing has changed except that 1 has been added to the store. Likewise, a sensible result — expressing the fact that both 1 and 2 have been added to the store, but nothing has happened to 3 — is derived when the *just* operator is applied to the event $add1\_\&\_add2$, defined as the simultaneous conjunction of the loosely defined events $add1$ and $add2$:

$$add1\_\&\_add2 = add1 \cap add2 = \{\, (s, s') \in W \times W \mid 1 \notin s \wedge 2 \notin s \wedge 1 \in s' \wedge 2 \in s' \,\}$$
$$consistent(add1\_\&\_add2) = central(add1\_\&\_add2) = \{n_\varnothing, n_{\{3\}}\}$$
$$\begin{aligned} & just(add1\_\&\_add2) \\ &= \{\, (s, s') \in add1\_\&\_add2 \mid \forall\, n \in central(add1\_\&\_add2) \bullet (s, s') \in n \,\} \\ &= \{\, (s, s') \mid 1 \notin s \wedge 2 \notin s \wedge 1 \in s' \wedge 2 \in s' \wedge \forall\, n \in \{n_\varnothing, n_{\{3\}}\} \bullet (s, s') \in n\} \\ &= \{\, (s, s') \mid 1 \notin s \wedge 2 \notin s \wedge 1 \in s' \wedge 2 \in s' \wedge s \cap \{3\} = s' \cap \{3\}\} \\ &= \{\, (\varnothing, \{1, 2\}), (\{3\}, \{1, 2, 3\}) \,\} \end{aligned}$$

However, there are cases in which central neutrals do not handle the frame problem satisfactorily. Consider the events $keepout1$ and $add1\_or\_keepout1$, defined as follows (where set union is used to represent simultaneous disjunction). Intuitively, if $add1\_or\_keepout1$ alone occurs in a given state, the membership status of 2 and 3 should not change, but there should be two alternative poststates — one in which 1 has become a member, and one in which it has not. However, it transpires that every neutral is both consistent with and central to this event, because it allows poststates in which nothing has changed at all — an outcome compatible with the simultaneous satisfaction of all the neutrals. Thus, contrary to intuition, from a given prestate (in which 1 is not a member), the only poststate allowed by $just(add\_or\_keepout1)$ is identical to the prestate.

$$keepout1 = \{\, (s, s') \in W \times W \mid 1 \notin s \wedge 1 \notin s' \,\}$$
$$add1\_or\_keepout1 = add1 \cup keepout1 = \{\, (s, s') \in W \times W \mid 1 \notin s \wedge (1 \in s' \vee 1 \notin s') \,\}$$
$$consistent(add1\_or\_keepout1) = central(add1\_or\_keepout1) = N$$
$$\begin{aligned} & just(add1\_or\_keepout1) \\ &= \{\, (s, s') \in add1\_or\_keepout1 \mid \\ &\qquad \forall\, n \in central(add1\_or\_keepout1) \bullet (s, s') \in n \,\} \\ &= \{\, (s, s') \in add1\_or\_keepout1 \mid \forall\, n \in N \bullet (s, s') \in n \,\} \\ &= \{\, (s, s') \in add1\_or\_keepout1 \mid s = s' \,\} \\ &= \{\, (s, s') \mid 1 \notin s \wedge s = s' \,\} \\ &= \{\, (\varnothing, \varnothing), (\{2\}, \{2\}), (\{3\}, \{3\}), (\{2, 3\}, \{2, 3\}) \,\} \end{aligned}$$

Another problematic example is the event $add1\_or\_add2$, defined as follows, which allows one or both of 1 and 2 to become members of the store:

$$\begin{aligned} & add1\_or\_add2 = add1 \cup add2 \\ &= \{\, (s, s') \in W \times W \mid (1 \notin s \wedge 1 \in s') \vee (2 \notin s \wedge 2 \in s') \,\} \end{aligned}$$
$$consistent(add1\_or\_add2) = \{\, n_\varnothing, n_{\{1\}}, n_{\{2\}}, n_{\{3\}}, n_{\{1,3\}}, n_{\{2,3\}} \,\}$$
$$central(add1\_or\_add2) = \{n_\varnothing, n_{\{3\}}\}$$
$$\begin{aligned} & just(add1\_or\_add2) \\ &= \{\, (s, s') \in add1\_or\_add2 \mid \forall\, n \in central(add1\_or\_add2) \bullet (s, s') \in n \,\} \\ &= \{\, (s, s') \in add1\_or\_add2 \mid (s, s') \in n_\varnothing \wedge (s, s') \in n_{\{3\}} \,\} \\ &= \{\, (s, s') \in add1\_or\_add2 \mid s \cap \varnothing = s' \cap \varnothing \wedge s \cap \{3\} = s' \cap \{3\} \,\} \\ &= \{\, (s, s') \in add1\_or\_add2 \mid s \cap \{3\} = s' \cap \{3\} \,\} \\ &= \{\, (s, s') \mid ((1 \notin s \wedge 1 \in s') \vee (2 \notin s \wedge 2 \in s')) \wedge s \cap \{3\} = s' \cap \{3\} \,\} \\ &= \{\, (\varnothing, \{1\}), (\varnothing, \{2\}), (\varnothing, \{1, 2\}), (\{1\}, \{1, 2\}), (\{1\}, \{2\}), \\ &\qquad (\{2\}, \{1, 2\}), (\{2\}, \{1\}), (\{3\}, \{1, 3\}), (\{3\}, \{1, 2, 3\}), \\ &\qquad (\{1, 3\}, \{2, 3\}), (\{1, 3\}, \{1, 2, 3), (\{2, 3\}, \{1, 3\}), (\{2, 3\}, \{1, 2, 3\})\} \end{aligned}$$

In this case — in contrast to the previous one — the result is not too restrictive, but too lax; for the central neutrals fix only the membership status of 3, thereby allowing the state pairs $(\{2\}, \{1\})$ and $(\{1\}, \{2\})$, in which one of 1 and 2 has been added, but the other has been removed at the same time.

Informally speaking, both the problem cases considered seem to involve events which are in some sense "intrinsically non-deterministic", even when "the rest stays the same". For a given system (W, E, N, C), let a *basic invariant* b be a subset of W which bisects W (assuming W is finite), i.e. such that $\mid b \mid = \mid W \setminus b \mid$.

Then a basic event could be defined as any event which takes one of the forms

$$\{ (s, s') \mid s \in b \} \quad \{ (s, s') \mid s \notin b \} \quad \{ (s, s') \mid s' \in b \} \quad \{ (s, s') \mid s' \notin b \}$$

for any basic invariant b — in which case events such as $add1$, $keepout1$, etc., are conjunctions of basic events, because invariants such as $1 \notin s$ are basic. Using this terminology (not used by Pitt and Byers), the problematic events $add1\_or\_keepout1$ and $add1\_or\_add2$ are both equivalent to disjunctions of two distinct conjunctions of basic events, which suggests a possible general form for such problematic cases.

In [PiB94], Pitt and Byers also define *relational/neutral systems with an invariant* (in which system states must satisfy an invariant, preserved by events), and consider how concurrency and the frame problem could be handled in models involving states persisting over finite intervals and events of finite duration.

2.3. Some Other Related Work

Early work on the frame problem in the artificial intelligence literature, e.g. by McCarthy and Hayes [McH69], is in the context of the *situation calculus*, an approach used to formalise the planning of sequences of actions to achieve goals, in simple domains such as the "blocks world". "Actions" (specified in terms of preconditions and postconditions) and "situations" (regarded as cumulative histories rather than states) are represented by first-order terms in a sorted theory in second-order predicate calculus. Predicates and functions whose extensions can change over time — so-called *fluents* — take a "situation" as one of their arguments. In the original situation calculus, actions are deterministic and instantaneous, and simultaneous actions are not considered. A function *do* is defined such that the term $do(a, s)$ denotes the situation arrived at if the action denoted by a is performed in the situation denoted by s. The frame problem arises here as the problem of generating appropriate explicit "frame axioms" in order to determine whether sentences true in situation s — involving fluents — remain true in situation $do(a, s)$, given a fairly loose pre/postcondition specification of the action a.

Work on the frame problem in variants of the situation calculus has continued. Reiter [Rei91] proposes a solution that works for deterministic actions, when state invariants do not need to be considered. Later papers in a broadly similar vein address non-deterministic actions [Lin96] and simultaneous (i.e. truly concurrent) actions [Rei96] — although apparently (unlike SSE), not both.

In [BMR95], Borgida, Mylopoulus, and Reiter describe the frame problem very clearly, citing both the situation calculus work, and that of Schuman, Pitt, and Byers. They list some of the ways in which existing specification languages express the changes associated with events, e.g. "writable" or "read only" state components (free to change, or not, respectively — used in VDM [Jon90]); "modifies at most" clauses, which may contain complex formulae (used in Larch [Gut85] and COLD [Jon91]), and the "Ξ" predicate constructor (used in Z [Spi88] to express in abbreviated notation a formula stating that certain specific state components do not change). Such features help to give more concise definitions of fairly simple individual operations, but cannot solve the frame problem in general — for example, they cannot handle simultaneous events. Borgida, Mylopoulus, and Reiter suggest a solution based on "explanation closure" axioms (which does not appear, however, to apply to simultaneous events), and give an algorithm for generating them in some cases. They also emphasize the relevance of the frame problem to the development of object-oriented specification languages, several of which have been proposed in the last decade or so [LaH94, Smi00]. In such languages, one class may inherit the state components and events (or "attributes" and "methods", in OO terminology) of another class, and add new state components, events, or state invariants of its own. The problem then arises of finding concise ways of expressing the effect of the inherited events on states of the subclass. Existing languages have some features that help to handle this — e.g. in Object-Z [Smi00], each event definition incorporates a "Δ-list", specifying which state components are allowed to change when that event occurs (any not listed being assumed not to change). However, such features have their limitations — for example, Δ-lists would not work in any model that allowed simultaneous events, because the Δ-list of one event would preclude any other simultaneous events from changing the state components listed.

3. The Semantics of Simultaneous Events (SSE)

The key concepts of SSE are now introduced in turn, and illustrated in relation to the number store described in section 1. The analysis is expressed in conventional logical and set notation. It should be emphasized that

SSE itself is not a formal language: it is rather, a semantic model of systems involving simultaneous events — incorporating a mechanism for handling the frame problem — which could serve as the semantic foundation for a formal specification language.

3.1. Generic Definitions

Conjunction, disjunction, and negation are defined as generic operators (applied later to *events* and *invariants*) on sets of an arbitrary type X, in terms of intersection, union, and complementation. $\underline{xor}$ (exclusive disjunction) and $\underline{imp}$ (implication) are also used, as binary operators.

$$
\begin{array}{|l}
[X] \\
\hline
(\_\ \underline{\&}\ \_), (\_\ \underline{or}\ \_), (\_\ \underline{xor}\ \_), (\_\ \underline{imp}\ \_) : \mathbb{P}\,X \times \mathbb{P}\,X \rightarrow \mathbb{P}\,X \\
\underline{not} : \mathbb{P}\,X \rightarrow \mathbb{P}\,X \\
\&, or, not : \mathbb{P}\,\mathbb{P}\,X \rightarrow \mathbb{P}\,X \\
\hline
\forall\, a, b : \mathbb{P}\,X \bullet (a\ \underline{\&}\ b) = a \cap b \wedge (a\ \underline{or}\ b) = a \cup b \\
\forall\, a : \mathbb{P}\,X \bullet \underline{not}(a) = \mathbb{P}\,X \setminus a \\
\forall\, a, b : \mathbb{P}\,X \bullet a\ \underline{xor}\ b = (a\ \underline{\&}\ \underline{not}(b))\ \underline{or}\ (\underline{not}(a)\ \underline{\&}\ b) \\
\forall\, a, b : \mathbb{P}\,X \bullet a\ \underline{imp}\ b = (\underline{not}(a))\ \underline{or}\ b \\
\forall\, A : \mathbb{P}\,\mathbb{P}\,X \bullet \&(A) = \bigcap A \wedge or(A) = \bigcup A \wedge not(A) = \underline{not}(\&(A))
\end{array}
$$

For an arbitrary type X, the sets of all finite sequences, all infinite sequences, and all finite or infinite sequences, are defined as follows:

$$
\begin{array}{|l}
[X] \\
\hline
\mathrm{anyseq}\ X == \\
\quad \{ f : \mathbb{N}_1 \rightarrowtail\!\!\!\!\!\!\shortmid\ \ X \mid \forall\, n : \mathbb{N}_1 \bullet n + 1 \in \mathrm{dom}\, f \Rightarrow n \in \mathrm{dom}\, f \} \\
\mathrm{finseq}\ X == \{ seq \in \mathrm{anyseq}\ X \mid \exists\, n : \mathbb{N}_1 \bullet n \notin \mathrm{dom}\, seq \} \\
\mathrm{infseq}\ X == \mathrm{anyseq}\ X \setminus \mathrm{finseq}\ X
\end{array}
$$

3.2. State Components, Types, and Values

COMP is a given set of *state components* (not those of a particular system, but rather a "universal" set of state components). Each component has a *type*, specified by the function *type*.

$$
\mid\ type : COMP \rightarrow TYPE
$$

Types are assumed to be non-empty disjoint sets of *values*. The set $TYPE$ of all types includes an arbitrary finite number of "basic" types, which are sets of "basic" values, and all higher-order types (function, powerset, and Cartesian product types) constructed recursively from those basic types. *VAL*, the set of all values, includes not only basic values, but also the higher-order entities (functions, sets of sets, and sets of tuples) from which the higher types are composed.

Throughout section 3, the examples involve only basic-typed state components. However, higher-typed components, and the implications of their use for the handling of the frame problem, will be examined further in section 4.

3.3. States and State Invariants

A *state* is a partial function whose domain is some non-empty subset of the universal set *COMP* of state components, and whose range is a set of values, each appropriately typed for its corresponding state component. (N.B. States involving infinite sets of components are not excluded here, because there will be a need to use such states in section 4.) $STATE$ is the set of all states:

$$STATE == \{\, s : COMP \rightarrow\!\!\!\!\!+ \; VAL \mid \mathrm{dom}\, s \neq \varnothing \wedge \forall\, c : COMP \bullet c \in \mathrm{dom}\, s \Rightarrow s(c) \in type(c) \,\}$$

For the sake of readability in later axioms, it is useful to define $comps(s)$ as the set of state components involved in state s, and $states(C)$ as the set of all states of which C is the set of components.

$$\begin{array}{|l}
comps : STATE \rightarrow \mathbb{P}\, COMP \\
states : \mathbb{P}\, COMP \rightarrow \mathbb{P}\, STATE \\
\hline
\forall\, s : STATE \bullet comps(s) = \mathrm{dom}\, s \\
\forall\, C : \mathbb{P}\, COMP \bullet states(C) = \{\, s : STATE \mid comps(s) = C \,\}
\end{array}$$

INV, the set of all *state invariants*, is the powerset of the set of all states (thus an invariant is any set of states). $true$, another name for the set of all states, is an invariant satisfied by all states. For any component c and value v, the invariant $val(c, v)$ is the set of all states in which c is involved and has the value v.

$$INV == \mathbb{P}\, STATE$$

$$\begin{array}{|l}
true : INV \\
val : COMP \times VAL \rightarrow INV \\
\hline
true = STATE \\
\forall\, c : COMP;\ v : VAL \bullet val(c, v) = \{\, s : STATE \mid c \in comps(s) \wedge s(c) = v \,\}
\end{array}$$

Example 3.1. The number store of section 1 could be modelled using three distinct state components of a (basic) type $Bool$:

$$\begin{array}{|l}
Bool : TYPE;\ T, F : VAL \\
in1, in2, in3 : COMP \\
\hline
Bool = \{T, F\} \wedge T \neq F \\
\#\{in1, in2, in3\} = 3 \wedge type(in1) = type(in2) = type(in3) = Bool
\end{array}$$

In that case, the following set of pairs is a state, in which the numbers 1 and 2 are in the number store, but 3 is not:

$$\begin{array}{|l}
IN_{\{1,2\}} : STATE \\
\hline
IN_{\{1,2\}} = \{in1 \mapsto T, in2 \mapsto T, in3 \mapsto F\}
\end{array}$$

An example of a state invariant involving the state components $in1$, $in2$, and $in3$ is:

$$\begin{array}{|l}
max_two : INV \\
\hline
max_two = or(\{val(in1, F), val(in2, F), val(in3, F)\})
\end{array}$$

This invariant is satisfied by any state in which at least one of $in1$, $in2$, and $in3$ is involved and has the value F. Regarding states of which $in1$, $in2$, and $in3$ are the only components, max_two means that the store can hold at most two of the three numbers 1, 2, and 3 at the same time.

3.4. Events

Events (types of occurrence) are sets of ordered pairs $(inv, \langle s, s' \rangle)$, where $\langle s, s' \rangle$ is a sequence of two states (prestate and poststate) satisfying the invariant inv, and involving the same state components. $EVENT$ is the set of all events.

$$EVENT == \mathbb{P}\{\, (inv, \langle s, s' \rangle) : INV \times finseqSTATE \mid \{s, s'\} \subseteq inv \wedge comps(s) = comps(s') \,\}$$

Pairs $(inv, \langle s, s' \rangle)$ are preferred to triples (inv, s, s') because they suggest a generalisation to events consisting of pairs in which the second element is a sequence of two or more states. (A related idea — in the context of

VDM-style specification — is applied in [Bic95] to the semantics of refining a pre/postcondition operation into implementations involving intermediate states.) However, such a generalised semantics, involving non-instantaneous events, would require a more complex mechanism for handling the frame problem, and is not explored further here.

Although the states s and s' in a given pair $(inv, \langle s, s' \rangle)$ belonging to an event have the same state components, different pairs belonging to the same event may involve states with different sets of state components. Thus SSE events are not required to be defined solely in terms of the states of one particular set of state components — some happen to have that property, but most do not. This contrasts with the events of simple state-transition systems (see section 2.1), which are binary relations on a particular state space; and with VDM-style operations with "read/write frames" (see [Bic95]), each of which specifies various restrictions on a particular set of state components. Likewise, different pairs $(inv, \langle s, s' \rangle)$ belonging to a given SSE event are not required to involve the same invariant inv. Thus SSE events are in general very loosely defined, independently of any particular set of state components or any particular invariant. The incorporation of invariants into events themselves means that their behaviour under various invariants is already implicit in their own internal semantic structure.

The events $preval(c, v)$ and $postval(c, v')$ occur when v and v' are the values of state component c in the prestate s, and poststate s', respectively (neither event constrains what other state components are present, or what happens to them). If v and v' do not belong to $type(c)$, there are no such prestates or poststates, and both $preval(c, v)$ and $postval(c, v')$ evaluate to the empty event $\varnothing$. For the sake of brevity in later axioms, it is useful to define the event $trans(c, v, v')$ (a transition for state component c) as the conjunction of $preval(c, v)$ and $postval(c, v')$.

$$
\begin{array}{|l}
preval, postval : COMP \times VAL \rightarrow EVENT \\
trans : COMP \times VAL \times VAL \rightarrow EVENT \\
\hline
\forall\, c : COMP, v, v' : VAL \bullet \\
\quad preval(c, v) = \{ (inv, \langle s, s' \rangle) \in INV \times \text{finseq}STATE \mid \{s, s'\} \subseteq inv \\
\qquad \wedge\ comps(s) = comps(s') \wedge c \in comps(s) \wedge s(c) = v \} \wedge \\
\quad postval(c, v') = \{ (inv, \langle s, s' \rangle) \in INV \times \text{finseq}STATE \mid \{s, s'\} \subseteq inv \\
\qquad \wedge\ comps(s) = comps(s') \wedge c \in comps(s) \wedge s'(c) = v' \} \wedge \\
\quad trans(c, v, v') = preval(c, v)\ \underline{\&}\ postval(c, v')
\end{array}
$$

Example 3.2. Given the states of the number store, as defined in *Example* 3.1, the following (and similar events $prein2$, $prein3$, etc. involving the state components $in2$ and $in3$) are examples of events of the forms $preval(c, v)$ and $postval(c, v')$:

$$
\begin{array}{|l}
prein1, preout1, postin1, postout1 : EVENT \\
\hline
prein1 = preval(in1, T) \wedge preout1 = preval(in1, F) \\
postin1 = postval(in1, T) \wedge postout1 = postval(in1, F)
\end{array}
$$

Additions and removals of individual numbers from the store (analogous to some of those discussed in section 2 in connection with the Pitt–Byers model), and events that keep an individual number in the store, or keep it out, can be modelled as conjunctions of such events, e.g.:

$$
\begin{array}{|l}
add1, rem1, keepin1, keepout1 : EVENT \\
\hline
add1 = preout1\ \underline{\&}\ postin1 \wedge rem1 = prein1\ \underline{\&}\ postout1 \\
\wedge\ keepin1 = prein1\ \underline{\&}\ postin1 \wedge keepout1 = preout1\ \underline{\&}\ postout1
\end{array}
$$

Similar definitions are assumed for $add2, \ldots, add3, \ldots$, etc. All of them are loosely defined, e.g. $add1$ can occur simultaneously with $add2$ or $rem2$ (although not with $rem1$), in a wide variety of systems, with various invariants and sets of state components, as long as $in1$ is one of the components present. The two disjunctive events found to be problematical for the Pitt–Byers model can be defined in terms of such events as ($add1$ $\underline{or}$ $add2$) and ($add1$ $\underline{or}$ $keepout1$).

3.5. Frame Problem Operators

SSE has four "frame problem" operators — *any_trans*, *fix*, $just_W$ ("weak just"), and $just_S$ ("strong just"). The last two provide alternative ways — the second more stringent than the first — of restricting the state changes associated with the occurrence of an event or set of simultaneous events.

For any set C of state components, $any\_trans(C)$ occurs when the prestate and poststate components include the members of C (possibly with other components as well) — whatever their prestate and poststate values; and $fix(C)$ occurs when the components in C are included, and do not change (i.e. they all make transitions to poststate values identical to their prestate values).

$$\begin{array}{|l}
any_trans, fix : (\mathbb{P}\, COMP) \rightarrow EVENT \\
\hline
\forall\, C : \mathbb{P}\, COMP \bullet any_trans(C) = \\
\qquad \&(\{\, or\{\, trans(c, v, v') \mid \{v, v'\} \subseteq type(c) \,\} \mid c \in C \,\}) \\
\forall\, C : \mathbb{P}\, COMP \bullet fix(C) = \\
\qquad \&(\{\, or\{\, trans(c, v, v) \mid v \in type(c) \,\} \mid c \in C \,\})
\end{array}$$

Example 3.3. The events (a) $add1$, (b) $add1 \;\underline{\&}\; fix(\{in2, in3\})$, and (c) $add1 \;\underline{\&}\; any\_trans(\{in2, in3\})$, are subtly different. (N.B. See *Example* 3.2 for the definition of $add1$.) Semantically speaking, (b) and (c) consist of pairs $(inv, \langle s, s' \rangle)$ belonging to (a), in which s and s' involve all three state components $in1$, $in2$, and $in3$ (and possibly others as well), whereas (a) also includes pairs in which $in2$ and/or $in3$ are not amongst the components of s and s'. The event (b) is a subset of (c), consisting of pairs in which both s and s' involve $in1$, $in2$, and $in3$ (possibly with other state components), and $in2$ and $in3$ have the same values in s' as in s.

The semantic differences between the events in *Example* 3.3 lead to behavioural differences when they occur in isolation, as will be shown in *Example* 3.7. However, before such differences can be understood, it is necessary to define the events $just_W(C, e)$ and $just_S(C, e)$, for any set C of state components, and any event e. Informally:

> $just_W(C, e)$ *is that event equivalent to an occurrence of* e*, but with no other changes to any members of* C *that are present in the prestate, beyond those changes associated with* e *or minimally required to preserve the prevailing state invariant.*
>
> $just_S(C, e)$ *is that event equivalent to an occurrence of* e*, with no other changes at all to any members of* C *that are present in the prestate.*

Both events allow any components of the prestate that are not in C to change in any way consistent with the occurrence of e and preservation of the prevailing state invariant. The "weak" event $just_W(C, e)$ has the same prestates as e, but restricts the set of poststates reachable from any given prestate. The "strong" event $just_S(C, e)$ — a further restriction of $just_W(C, e)$ — cuts out those cases in which an occurrence of e would imply "implicit" simultaneous changes to preserve the prevailing state invariant. The exclusion of such cases may involve restricting the set of prestates, and further restricting the set of poststates reachable from any given prestate.

The two "just" operators can be applied to a set E of simultaneous events, by using the conjunctive event $\&(E)$ as the second argument. The "strong" operator $just_S$ is defined in terms of the "weak" operator $just_W$. Before the latter can be defined formally, some auxiliary definitions are required. A *sufficient post event set* E for an event e in a state s is a set of non-empty events of the form $postval(c, v')$ (see section 3.4), whose conjunction $\&(E)$ is non-empty, and whose occurrence is sufficient to ensure the occurrence of e, in the prestate s; a *minimal sufficient post event set* E for e in s is a sufficient post event set E for e in s, with no proper subset that is; and a *key sufficient post event set* E for e in s is any union of one or more minimal sufficient post event sets for e in s, whose conjunction $\&(E)$ is non-empty. Such key sufficient post event sets are crucial for handling the disjunctive events problematic for the Pitt–Byers semantics.

$$
\begin{array}{|l}
(\_ \ \mathit{suff\_post\_set\_for} \ \_ \ \mathit{in} \ \_) : \mathbb{P}((\mathbb{P}\,\mathit{EVENT}) \times \mathit{EVENT} \times \mathit{STATE}) \\
(\_ \ \mathit{min\_suff\_post\_set\_for} \ \_ \ \mathit{in} \ \_) : \mathbb{P}((\mathbb{P}\,\mathit{EVENT}) \times \mathit{EVENT} \times \mathit{STATE}) \\
(\_ \ \mathit{key\_suff\_post\_set\_for} \ \_ \ \mathit{in} \ \_) : \mathbb{P}((\mathbb{P}\,\mathit{EVENT}) \times \mathit{EVENT} \times \mathit{STATE}) \\
\hline
\forall E : \mathbb{P}\,\mathit{EVENT};\ e : \mathit{EVENT};\ s : \mathit{STATE} \bullet \\
\quad (E \ \mathit{suff\_post\_set\_for} \ e \ \mathit{in} \ s) \Leftrightarrow \varnothing \subset \&(E) \\
\qquad \wedge (\forall e' \in E \bullet \exists c : \mathit{COMP};\ v' : \mathit{VAL} \bullet v' \in \mathit{type}(c) \wedge e' = \mathit{postval}(c, v')) \\
\qquad \wedge (\forall s' : \mathit{STATE} \bullet (\mathit{true}, \langle s, s' \rangle) \in \&(E) \Rightarrow (\mathit{true}, \langle s, s' \rangle) \in e) \\
\forall E : \mathbb{P}\,\mathit{EVENT};\ e : \mathit{EVENT};\ s : \mathit{STATE} \bullet \\
\quad (E \ \mathit{min\_suff\_post\_set\_for} \ e \ \mathit{in} \ s) \Leftrightarrow \\
\qquad (E \ \mathit{suff\_post\_set\_for} \ e \ \mathit{in} \ s) \wedge \neg \exists E' \subset E \bullet (E' \ \mathit{suff\_post\_set\_for} \ e \ \mathit{in} \ s) \\
\forall E : \mathbb{P}\,\mathit{EVENT};\ e : \mathit{EVENT};\ s : \mathit{STATE} \bullet \\
\quad (E \ \mathit{key\_suff\_post\_set\_for} \ e \ \mathit{in} \ s) \Leftrightarrow \\
\qquad \&(E) \neq \varnothing \wedge \exists \mathit{Es} : \mathbb{P}(\mathbb{P}\,\mathit{EVENT}) \bullet \\
\qquad\quad E = \bigcup \mathit{Es} \wedge \forall E' \in \mathit{Es} \bullet (E' \ \mathit{min\_suff\_post\_set\_for} \ e \ \mathit{in} \ s)
\end{array}
$$

Example 3.4. In terms of the events defined in *Example* 3.2, the minimal sufficient post event sets for the event $e = (\mathit{add}1 \ \underline{\mathit{or}} \ \mathit{add}2)$, in the state $s = \{\mathit{in}1 \mapsto F, \mathit{in}2 \mapsto F, \mathit{in}3 \mapsto T\}$ (in which both 1 and 2 are out of the store), are $\{\mathit{postin}1\}$ (1 is in the store in the poststate) and $\{\mathit{postin}2\}$ (2 is in the store in the poststate). The key sufficient post event sets (consistent combinations of one or more minimal sufficient post event sets) for that s and e are thus $\{\mathit{postin}1\}$, $\{\mathit{postin}2\}$, and their union $\{\mathit{postin}1, \mathit{postin}2\}$.

Example 3.5. For the event $e = (\mathit{add}1 \ \underline{\mathit{or}} \ \mathit{keepout}1)$ from *Example* 3.2, in the state $s = \{\mathit{in}1 \mapsto F, \mathit{in}2 \mapsto F, \mathit{in}3 \mapsto T\}$ (in which 1 is out of the store), the minimal sufficient post event sets are $\{\mathit{postin}1\}$ (1 is in the store in the poststate) and $\{\mathit{postout}1\}$ (1 is out of the store in the poststate). These are also the only key sufficient post event sets, because their union is not consistent (in the poststate, 1 is either in the store or out, but not both).

Example 3.6. Consider the conditional event *condadd*, which does *add*2 or *add*3 according to whether 1 is in the store in the prestate, or not:

$$
\begin{array}{|l}
\mathit{condadd} : \mathit{EVENT} \\
\hline
\mathit{condadd} = (\mathit{prein}1 \ \underline{\mathit{or}} \ \mathit{preout}1) \ \underline{\&} \ (\mathit{prein}1 \ \underline{\mathit{imp}} \ \mathit{add}2) \ \underline{\&} \ (\mathit{preout}1 \ \underline{\mathit{imp}} \ \mathit{add}3)
\end{array}
$$

The first conjunct here ensures that *condadd* cannot happen in prestates whose components do not include *in*1 (because the definitions of both *prein*1 and *preout*1 imply that *in*1 is present). Thus for one of the two conditional conjuncts, it must be the case that both the "antecedent" and "consequent" events occur (and consequently, given the definitions of *add*2 and *add*3 in *Example* 3.2, the prestate and poststate must contain at least one of the components *in*2 and *in*3, as well as *in*1). The only "minimal sufficient post event set" for *condadd* in the prestate $s = \{\mathit{in}1 \mapsto T, \mathit{in}2 \mapsto F, \mathit{in}3 \mapsto F\}$ (in which 1 is in the store and 2 and 3 are out) is $\{\mathit{postin}2\}$ (2 is in the store in the poststate). This is also the sole "key sufficient post event set" for *condadd* in that s. In the prestate $s = \{\mathit{in}1 \mapsto F, \mathit{in}2 \mapsto F, \mathit{in}3 \mapsto F\}$ (in which 1, 2, and 3 are out of the store), there is again just one "minimal sufficient post event set" for *condadd*, which is also the sole "key sufficient post event set" — namely $\{\mathit{postin}3\}$ (3 is in the store in the poststate).

The definition of *condadd* is loose, with no explicit specification, in the consequent of each conditional branch, of what else changes (or not) concurrently. Such conditional events are identified in [BMR95] as being particularly awkward to handle as regards the frame problem. Nonetheless, as shown later in *Example* 3.11, *condadd* is correctly constrained when the two "just" operators are applied to it. (Of course, similar but more tightly defined events, which positively rule out simultaneous changes to certain state components, can be obtained by adding extra subformulae involving the previously defined *fix* operator.)

The event $\mathit{just}_W(C, e)$ is now defined as the set of all pairs $(\mathit{inv}, \langle s, s' \rangle)$ in e, that belong to $\&(E)$ for some key sufficient post event set E for e in the prestate s, such that no other pair $(\mathit{inv}, \langle s, s'' \rangle)$ in $\&(E)$ involves less change between prestate and poststate, regarding the state components in C. (N.B. The symbol $\lhd$ means domain restriction — hence $C \lhd (s \cap s')$ is the set of pairs $(\mathit{comp}, \mathit{val})$ belonging to both s and s', such that comp belongs to C.)

$$
\begin{array}{|l}
just_W : (\mathbb{P}\, COMP) \times EVENT \to EVENT \\
\hline
\forall\, C : \mathbb{P}\, COMP;\ e : EVENT \bullet just_W(C, e) = \\
\quad \{\, (inv, \langle s, s' \rangle) \in e \mid \exists\, E : \mathbb{P}\, EVENT \bullet \\
\qquad\qquad (E\ key\_suff\_post\_set\_for\ e\ in\ s) \wedge (inv, \langle s, s' \rangle) \in \&(E) \\
\qquad\qquad \wedge \neg\, (\exists\, s'' : STATE \bullet (inv, \langle s, s'' \rangle) \in \&(E) \\
\qquad\qquad\qquad\qquad\qquad \wedge\ C \lhd (s \cap s') \subset C \lhd (s \cap s'')) \,\}
\end{array}
$$

To preserve an invariant inv, going from s to s' may involve changing more state components than explicitly specified by the events in a given key sufficient post set E, but the condition $C \lhd (s \cap s') \subset C \lhd (s \cap s'')$ minimises any such extra changes relative to that C, inv, s and E. If inv is $true$ (see section 3.3), which is satisfied by any poststate s' whatsoever, there are guaranteed to be no extra changes.

The properties below provide an initial impression of the $just_W$ operator (proofs are omitted, because they are fairly trivial).

$$
\begin{array}{ll}
just_W(C, e) \subseteq e & (just_W(C, e) \text{ restricts } e) \\
just_W(C, just_W(C, e)) = just_W(C, e) & (just_W \text{ is a fixed-point op.}) \\
C_1 \subseteq C_2 \Rightarrow just_W(C_2, e) \subseteq just_W(C_1, e) & (\text{wider scope, more restriction}) \\
just_W(\varnothing, e) = e & (\text{empty scope, no restriction}) \\
C_1 \subseteq C_2 \Rightarrow & (\text{wider scope dominates}) \\
\quad just_W(C_1, just_W(C_2, e)) = just_W(C_2, e) & \\
just_W(C, (e_1\ \underline{or}\ e_2)) = & \\
\quad just_W(C, e_1)\ \underline{or}\ just_W(C, e_2)\ \underline{or}\ just_W(C, (e_1\ \underline{\&}\ e_2)) & \\
just_W(C, (e_1\ \underline{xor}\ e_2)) = just_W(C, e_1)\ \underline{xor}\ just_W(C, e_2) &
\end{array}
$$

Example 3.7. Let $C_{store} = \{in1, in2, in3\}$. The events obtained by applying $just_W$ to the events defined in *Example* 3.3, with respect to C_{store}, are:

(d) $just_W(C_{store}, add1)$

(e) $just_W(C_{store}, add1\ \underline{\&}\ fix(\{in2, in3\}))$

(f) $just_W(C_{store}, add1\ \underline{\&}\ any\_trans(\{in2, in3\}))$

Event (d) consists of pairs $(inv, \langle s, s' \rangle)$ in which $in1$ is present in s and s', and changes from F to T. If components $in2$ and $in3$ are present in s as well (which they may not be), they change *only* if preserving inv minimally requires them to (e.g. if inv is max_two from *Example* 3.1, and the store is full in s, holding both numbers 2 and 3, then (d) implies either a simultaneous $rem2$, or a simultaneous $rem3$ — but not both). Any other state components present in s and s' are free to change in any way consistent with preserving inv.

Event (e) is more restricted than (d). It contains pairs $(inv, \langle s, s' \rangle)$ such that $in1$, $in2$, and $in3$ are all present in s and s', $in1$ changes from F to T, and $in2$ and $in3$ do not change. This event cannot occur when inv is max_two, and the store holds both 2 and 3 in s. As with event (d), any components in s and s' which are not in C_{store} are left unconstrained by event (e), except that inv must be preserved.

As regards components in C_{store}, event (f) is less restricted than (d). It contains pairs $(inv, \langle s, s' \rangle)$ such that $in1$, $in2$, and $in3$ are all present in s and s', $in1$ changes from F to T, and anything can happen to $in2$ and $in3$, as long as inv is preserved. Again, any components in s not belonging to C_{store} are free to change, as long as inv is preserved.

Example 3.8. Consider the following events, in which $just_W$ is applied to the disjunctive events defined in *Example* 3.2:

(g) $just_W(C_{store}, (add1\ \underline{or}\ add2))$

(h) $just_W(C_{store}, (add1\ \underline{or}\ keepout1))$

As argued in section 2.2, their counterparts in the Pitt–Byers model behave counter-intuitively: that of (g) is too lax, because in *any* prestate in which just one of 1 and 2 is in the store, it allows one number to be removed as the other is added; and that of (h) is too restrictive, because it allows no change at all. By comparison, in a prestate in which both 1 and 2 are out of the store, (g) permits the addition of 1, 2, or 1 and

2 (cases corresponding to the three key sufficient post event sets in *Example* 3.4), but no other concurrent changes to the membership status of 1, 2, or 3, *unless* required to preserve a prevailing state invariant *inv*. Whilst (g) is more restrictive than its Pitt–Byers counterpart, regarding additions and removals of 1, 2, and 3, (h) is more liberal — given a prestate in which 1 is out of the store, it allows 1 to be added to or kept out of the store (cases corresponding to the two key sufficient post event sets in *Example* 3.5), with no other concurrent additions or removals, beyond any minimally required to preserve a prevailing state invariant.

Example 3.9. Neither (g) nor (h) constrain state components which are not members of C_{store}. The following much stronger events (i) and (j) constrain all state components whatsoever, i.e. all members of the "universal" set $COMP$:

(i) $just_W(COMP, (add1 \ \underline{or} \ add2))$
(j) $just_W(COMP, (add1 \ \underline{or} \ keepout1))$

The use of $COMP$ here does not mean that all members of $COMP$ not in C_{store} must be present in a prestate s and a poststate s' for events (i) and (j) to occur — but it does mean that any that are present are prevented from changing unless required to in order to preserve a prevailing state invariant.

To obtain the events that are like (g) and (h) in *Example* 3.8, except that they do not allow *any* implicit simultaneous changes to preserve a state invariant, as regards state components in C_{store}, the "weak" $just_W$ is replaced by the "strong" $just_S$. For any component set C and event e, $just_S(C, e)$ is defined as the set of all pairs $(inv, \langle s, s' \rangle)$ belonging to $just_W(C, e)$, such that the pair $(true, \langle s, s' \rangle)$ — involving the same two states — belongs to $just_W(C, \&(E))$, where E is any key sufficient post event set for e in the prestate s.

$$
\begin{array}{l}
just_S : (\mathbb{P}\, COMP) \times EVENT \rightarrow EVENT \\
\hline
\forall\, C : \mathbb{P}\, COMP;\ e : EVENT \bullet just_S(C, e) = \\
\quad \{\, (inv, \langle s, s' \rangle) \in just_W(C, e) \mid \\
\qquad \exists\, E : \mathbb{P}\, EVENT \bullet (E \ key\_suff\_post\_set\_for \ e \text{ in } s) \\
\qquad\quad \wedge\ (true, \langle s, s' \rangle) \in just_W(C, \&(E)) \,\}
\end{array}
$$

Example 3.10. To help explain the definition of $just_S$, consider the "strong" analogue (k) of the event (g) in *Example* 3.8:

(k) $just_S(C_{store}, (add1 \ \underline{or} \ add2))$

Consider a prestate $s = \{in1 \mapsto F, in2 \mapsto F, in3 \mapsto T\}$ (in which 1 and 2 are out of the store, and 3 is in). Substituting into the definition of $just_S$, e is $(add1 \ \underline{or} \ add2)$, and the only possible key sufficient post event sets E for e in s are those given in *Example* 3.4, i.e. $\{postin1\}$, $\{postin2\}$, and $\{postin1, postin2\}$. Taking the first of these as an example, $just_W(C_{store}, \&(\{postin1\}))$ consists of pairs $(inv, \langle s, s' \rangle)$ in which the component $in1$ has the value T in s' (and the value T or F in s), and no other state components belonging to C_{store} that are present in s change in going from s to s', unless needed to preserve the invariant inv. The pairs belonging to this event involve many different invariants and sets of state components, but in those pairs where the invariant is $true$, there are guaranteed to be no extra changes needed, because $true$ is satisfied by any poststate s' whatsoever. Thus pairs $(inv, \langle s, s' \rangle)$ in $just_W(C_{store}, (add1 \ \underline{or} \ add2))$, where $s = \{in1 \mapsto F, in2 \mapsto F, in3 \mapsto T\}$, such that $(true, \langle s, s' \rangle)$ belongs to $just_W(C_{store}, \&(\{postin1\}))$, are guaranteed to be ones in which — regardless of what inv is — s and s' are identical apart from the change to the component $in1$. In other words, 1 will be added to the store, but nothing will change as regards the membership status of 2 and 3. For the same prestate s, the other two key sufficient post event sets — $\{postin2\}$, and $\{postin1, postin2\}$ — correspond to adding 2 to the store without changing anything else, and adding both 1 and 2 without moving 3.

Unlike its "weak" analogue (i) in *Example* 3.9, event (k) cannot occur when the prevailing invariant inv is max_two, and 2 and 3 are in the store in the prestate. This is because if the store is already at a maximum capacity of two in s, 1 could not be added without an "implicit" simultaneous removal (of 2 or 3) to preserve max_two in s' — but such extra changes are not allowed by $just_S$.

Example 3.11. Consider the events (l) and (m), where $condadd$ is the conditional event defined in *Example*

3.6:

(l) $just_W(C_{store}, condadd)$

(m) $just_S(C_{store}, condadd)$

The possible prestates of *condadd* can be divided into those in which $in1$ has the value T and $in2$ has the value F (i.e. 1 is in the store and 2 is out), and those in which $in1$ has the value F and $in3$ has the value F (i.e. both 1 and 3 are out of the store). For prestates s of these two kinds, the only possible key sufficient post event sets for *condadd* are $\{postin2\}$ and $\{postin3\}$, respectively. Thus (l) consists of all pairs $(inv, \langle s, s' \rangle)$, belonging to the loosely defined *condadd*, such that either 1 is in the store in s and 2 is added, or 1 is out of the store in s and 3 is added — and (in either case), nothing else changes *unless* necessary to preserve inv. For example, if inv were max_two from *Example* 3.1, s contained all three of the state components $in1$, $in2$, and $in3$, and both 1 and 3 were in the store in s, then adding 2 would imply a simultaneous removal of either 1 or 3 (but not both). The event (m) is a more restrictive event that definitely rules out *any* such implicit extra changes — when (m) occurs, and s contains all three of $in1$, $in2$, and $in3$ as state components, then either 2 is added to the store, 1 stays in, and 3 stays where it was in the prestate; or 3 is added, 1 stays out, and 3 stays where it was in the prestate. Thus the precondition of (m) is stricter than that of (l), because it must exclude cases in which additional implicit changes would be needed to preserve the prevailing invariant inv. Neither (l) nor (m) impose restrictions on any state components present in s and s' which are not members of C_{store}, other than requiring their behaviour to be consistent with preserving the prevailing state invariant inv, given any changes involving the components $in1$, $in2$, and $in3$.

Events (l) and (m) correctly model (according to the “weak” and “strong” interpretations, respectively) what happens when *condadd* occurs on its own, despite its loose definition. Interestingly, an occurrence of *condadd* does not exclude a simultaneous occurrence of $add2$ from taking place in a prestate that satisfies the precondition of $add2$ — the event $(condadd \mathbin{\underline{\&}} add2)$ is one in which 2 is added to the store regardless, and 3 is added as well if 1 is out of the store in the prestate. The behaviour of this event in isolation is also correctly constrained by the two “just” operators.

3.6. Histories and History Invariants

Informally speaking, a *history* is a (finite or infinite) non-empty labelled transition sequence, satisfying certain constraints, of the form

$$s_1 \xrightarrow{E_1} s_2 \xrightarrow{E_2} s_3 \ldots$$

where $E_1, E_2, \ldots$ and $s_1, s_2, \ldots$ are sequences of respectively, event sets, and states of a particular set of state components. Thus SSE can represent concurrency in terms of simultaneous occurrences of events, rather than in terms of interleaving, as is the case in typical previous state-based models of concurrency (see e.g. [CaC96], [Col97]).

Formally, a history is a 3-tuple (C, s^*, E^*), where s^* is a sequence of one or more states whose set of components is the non-empty set C, and E^* is a sequence of event sets which label the transitions between the states. (Here, $s^*(1), s^*(2), \ldots$ and $E^*(1), E^*(2), \ldots$ correspond to $s_1, s_2, \ldots$ and $E_1, E_2, \ldots$ respectively in the informal diagram just shown.) Two sets of histories are defined — the “weak” set $HIST_W$ and the “strong” set $HIST_S$, in which state transitions are governed by $just_W$ and $just_S$, respectively. $HIST_W$ is defined as follows:

$$\begin{aligned}
&HIST_W == \{ (C, s^*, E^*) : \\
&\quad \mathbb{P}_1\, COMP \times \mathrm{anyseq}\, states(C) \times \mathrm{anyseq}\, \mathbb{P}\, EVENT \mid \\
&\qquad 1 \in \mathrm{dom}\, s^* \wedge (\forall\, n : \mathbb{N}_1 \bullet n \in \mathrm{dom}\, E^* \Leftrightarrow n + 1 \in \mathrm{dom}\, s^*) \\
&\qquad \wedge \exists\, inv : INV \bullet (\mathrm{ran}\, s^*) \subseteq inv \wedge \forall\, n : \mathbb{N}_1 \bullet n \in \mathrm{dom}\, E^* \Rightarrow \\
&\qquad\quad (E^*(n) = \varnothing \wedge s^*(n+1) = s^*(n)) \vee (inv, \langle s^*(n), s^*(n+1) \rangle) \in just_W(C, \&(E^*(n))) \}
\end{aligned}$$

The first two top-level conjuncts here ensure that s^* is non-empty, and either both s^* and E^* are denumerably infinite, or both are finite and s^* contains one more element than E^*, so that the history ends in a state. The third top-level conjunct requires all states in s^* to satisfy some invariant inv, such that either the nth event set E_n is empty and states s_n and s_{n+1} are identical, or $(inv, \langle s_n\, s_{n+1} \rangle)$ belongs to the restriction of the conjunction $\&(E_n)$ by $just_W$, relative to the set C of components. This ensures that s_{n+1} reflects any state changes relating to the events in E_n and any minimal extra changes implied to ensure that s_{n+1} preserves *inv*, but is otherwise the same as s_n.

$HIST_S$ is that subset of $HIST_W$ in which state transitions satisfy the requirements of $just_S$ as well as those of $just_W$.

$$\begin{aligned} HIST_S == \{\, (C, s^*, E^*) : HIST_W \mid \\ \exists\, inv : INV \bullet (\operatorname{ran} s^*) \subseteq inv \wedge \forall\, n : \mathbb{N}_1 \bullet n \in \operatorname{dom} E^* \Rightarrow \\ (E^*(n) = \varnothing \wedge s^*(n+1) = s^*(n)) \vee (inv, \langle s^*(n), s^*(n+1)\rangle) \in just_S(C, \&(E^*(n))) \,\} \end{aligned}$$

The members of $HIST_S$ can be described as WYSIWYG ("What You See Is What You Get") histories, in the sense that the only state component changes that occur are those associated with the events explicitly named in the state transition labels. By comparison, $HIST_W$ also includes histories in which there are implicit extra changes, not shown in the state transition labels, in order to preserve a state invariant.

$INFHIST_W$ and $INFHIST_S$ are the sets of all infinite "weak" and "strong" histories, respectively.

$$INFHIST_W == \{\, (C, s^*, E^*) : HIST_W \mid \operatorname{dom} s^* = \mathbb{N}_1 \,\}$$

$$INFHIST_S == \{\, (C, s^*, E^*) : HIST_S \mid \operatorname{dom} s^* = \mathbb{N}_1 \,\}$$

$HINV$, the set of all *history invariants*, is the power set of $INFHIST_W$. The members of a history invariant need not all share the same state components. The history invariant *all_infhists*, another name for $INFHIST_W$, is satisfied by any history, whereas *strong_hists*, another name for $INFHIST_S$, is satisfied only by strong histories.

$$HINV == \mathbb{P}\, INFHIST_W$$

$$\begin{array}{|l} all\_infhists, strong\_infhists : HINV \\ \hline all\_infhists = INFHIST_W \wedge strong\_infhists = INFHIST_S \end{array}$$

Because $HIST_S$ is a proper subset of $HIST_W$ (i.e. every history that satisfies the strong constraints of $just_S$ also satisfies the weaker constraints of $just_W$), some history invariants may in fact consist entirely of "strong" histories; and any invariant can be made "strong" by simply conjoining it with the invariant *strong_infhists*. Thus there is no need to distinguish by name sets of "strong" and "weak" invariants.

3.7. Systems

A *system* is a 5-tuple $(C, E, init, sinv, hinv)$, where C is a non-empty set of state components, E is a non-empty set of events, *init* and *sinv* are state invariants (for the initial state and for all states, respectively), and *hinv* is a history invariant.

$$SYS == \{\, (C, E, init, sinv, hinv) : (\mathbb{P}_1\, COMP) \times (\mathbb{P}_1\, EVENT) \times INV \times INV \times HINV \,\}$$

For a system $sys = (C, E, init, sinv, hinv)$, the sets of "weak" and "strong" infinite histories, $infhists_W(sys)$ and $infhists_S(sys)$ are all those weak and strong histories, respectively, that satisfy *hinv*, *init*, and *sinv*, and that involve only states in $states(C)$ and events in E. These two sets of histories provide two different interpretations of a system, suited to different styles of specification — a "strong" one in which every state component change must be specified explicitly, and a "weak" one in which minimal extra changes can be implied to preserve the state invariant *sinv*. If *sinv* is *true* (satisfied by all states whatsoever), the weak and strong sets of histories are identical.

$$\begin{array}{|l} infhists_W, infhists_S : SYS \rightarrow \mathbb{P}\, HIST_W \\ \hline \forall\, sys : SYS;\ C : \mathbb{P}_1\, COMP;\ E : \mathbb{F}_1\, EVENT; \\ init, sinv : INV;\ hinv : HINV \bullet sys = (C, E, init, sinv, hinv) \Rightarrow \\ \quad infhists_W(sys) = \{\, (C, s^*, E^*) \in hinv \mid \\ \qquad s^*(1) \in init \wedge \operatorname{ran} s^* \subseteq (sinv \cap states(C)) \wedge \bigcup \operatorname{ran} E^* \subseteq E \,\} \\ \quad \wedge\ infhists_S(sys) = infhists_W(sys) \cap INFHIST_S \end{array}$$

In some cases, one or both sets of histories may evaluate to the empty set, e.g. if *init* or *sinv* are not satisfiable by any of the states in $states(C)$. Typically, *init* and *sinv* are not subsets of $states(C)$, but are more general invariants, satisfied by states involving other sets of components as well. Likewise, *hinv* typically contains infinite histories involving other sets of components, not just those involving C. For reasons discussed later in section 4.3, C and E are allowed to be infinite.

The history invariant of a system can be used to express behavioural constraints beyond those implied by the definitions of the states and events of that system — e.g. "fairness" constraints [MaP92], applied to individual events or to event sets, or "evolution invariants" [Bic95]. In general, constraints on the co-occurrence of events can be modelled either by making certain events mutually exclusive by definition, or by defining events more loosely, but using *hinv* to exclude certain co-occurrences. Some states and events of a system may never occur in any of its histories. (If none of the events are usable, histories contain infinitely repeated initial state sets, linked by empty event sets.)

Example 3.12. The system *store*1 is a variant of the Pitt–Byers number store, using state components, events and invariants defined in preceding examples, in which the store is initially empty, and numbers can be added or removed (singly or concurrently). Here, the general state invariant and the history invariant are *true* (satisfied by all states) and *all_infhists* (satisfied by all infinite histories), respectively.

$$\begin{array}{|l} store1 : SYS \\ \hline store1 = (C_{store}, \{add1, add2, add3, rem1, rem2, rem3\}, \\ \qquad val(in1, F) \ \underline{\&}\ val(in2, F) \ \underline{\&}\ val(in3, F), true, all\_infhists) \end{array}$$

The "weak" and "strong" interpretations of this system are identical, because since *true* is satisfied by any poststate, extra implicit changes are never needed to preserve it. The state transitions of this system include the following one, in which 3 is in the store in the prestate, and 1 and 2 are added:

$$\{in1 \mapsto F, in2 \mapsto F, in3 \mapsto T\} \xrightarrow{\{add1, add2\}} \{in1 \mapsto T, in2 \mapsto T, in3 \mapsto T\}$$

The prestate and poststate here are related by $just_W(C_{store}, \&(\{add1, add2\})$, which does not allow $in3$ to change its value if the prevailing state invariant is *true*.

Example 3.13. Consider a system *store*2 like *store*1, but with the general state invariant *max_two* instead of *true*, so that the store can only hold two numbers at a time. The "weak" and "strong" interpretations of this system are different, because some state transitions are possible in "weak" histories but not in "strong" ones, e.g.:

$$\{in1 \mapsto F, in2 \mapsto F, in3 \mapsto T\} \xrightarrow{\{add1, add2\}} \{in1 \mapsto T, in2 \mapsto T, in3 \mapsto F\}$$

Here, both *add*1 and *add*2 are enabled in the prestate, but their co-occurrence in a "weak" history requires an implicit *rem*3, to preserve the prevailing invariant *max_two*. (If the store were empty in the prestate, then adding 1 and 2 would not change *in*3.) In this case, a transition label $\{add1, add2, rem3\}$ would have the same effect, because the invariant and the prestate together imply a simultaneous *rem*3.

Example 3.14. Let *store*3 be a system like *store*1, but with the event ($add1$ $\underline{or}$ $add2$) instead of *add*1 and *add*2. This system behaves exactly the same as *store*1, in terms of the possible sequences of states — but unlike *store*1, it is non-deterministic, because ($add1$ $\underline{or}$ $add2$) sometimes allows more than one poststate.

4. Further Issues

This section discusses three further issues arising from SSE — the composition of systems, the modelling of events that have inputs and outputs, and the implications as regards the frame problem of the "granularity" of state components.

4.1. Composition of Systems

Composition of two systems is possible, in a very straightforward way, by taking the unions of the component and event sets of the subsystems, and conjoining their corresponding invariants. Such composition is self-evidently commutative and associative. (Other forms of composition could of course be defined — for example, an additional invariant might be imposed. What is defined here is the simplest conceivable form of composition.)

$$
\begin{array}{|l}
(\_ \,\underline{\|}\, \_) : SYS \times SYS \rightarrow SYS \\
\hline
\forall (C1, E1, init1, sinv1, hinv1), (C2, E2, init2, sinv2, hinv2) : SYS \bullet \\
\quad (C1, E1, init1, sinv1, hinv1) \,\underline{\|}\, (C2, E2, init2, sinv2, hinv2) = \\
\qquad (C1 \cup C2, E1 \cup E2, init1 \,\underline{\&}\, init2, sinv1 \,\underline{\&}\, sinv2, hinv1 \,\underline{\&}\, hinv2)
\end{array}
$$

In histories of the composed system, individual subsystems need not execute an event or set of events at every step. The most striking feature here is that no "promotion" of events is involved: the events of a typical SSE system — such as $add1$, $add2$, etc. from $store1$ in *Example* 3.12 — are defined so loosely, independently of any particular invariant or set of state components, that their behaviour in a context wider than $store1$ is already implicit in their semantic structure, and will be correctly constrained by the two "just" operators. This contrasts with conventional state-based approaches, in which events are defined as relations on the state space of a particular system, so that if a system is used as a part of a larger system, its events must be "promoted" into events on the state space of that larger system.

For the sake of clarity, SSE has been presented in terms of the very simple number store system used by Pitt and Byers. A more complex system, often referred to in the literature — is that of the "dining philosophers". One variant, described in [Hoa85], consists of five philosophers $PHIL_0, \ldots, PHIL_4$, each assigned a place at a circular table with corresponding forks $FORK_0, \ldots, FORK_4$, in anti-clockwise order. For any i from 0 to 4, let $i \oplus 1$ and $i \ominus 1$ denote $i + 1 \bmod 5$ and $i - 1 \bmod 5$, respectively. To eat some spaghetti from a bowl in the centre of the table, philosopher $PHIL_i$ must first sit down at the ith position, pick up fork i, pick up fork $i \oplus 1$, then (after eating, which is not explicitly modelled as an event) put down fork i, put down fork $i \oplus 1$, and finally get up. The events cannot be performed in any order other than that just given, although once standing up, a philosopher can go through the same cycle of events again.

The system can be composed from five philosopher systems and five fork systems, with some sharing of state components and events. Individual philosophers and forks can only execute one event at a time, but in the composed system, several events may occur simultaneously. Here, the ordering constraints on events are enforced by their preconditions — another approach would have been to define events less tightly, but to use history invariants to constrain their ordering. State invariants specify that philosophers cannot hold forks unless seated, and that forks cannot be held by two philosophers at the same time. Using i, $i \oplus 1$, and $i \ominus 1$ as meta-expressions, for which particular numbers should be substituted, and using the functions val from section 3.2, and $trans$ from section 3.3, the ith philosopher $PHIL_i$ is a tuple $(C_i, E_i, init_i, all_i, hinv_i)$, where:

$$
\begin{aligned}
&C_i = \{p_i\_has\_f_i, p_i\_has\_f_{i\oplus 1}, p_i\_seated\} \quad \text{(all of type } Bool\text{)} \\
&E_i = \{p_i\_sits\_down, p_i\_picks\_up\_f_i, p_i\_picks\_up\_f_{i\oplus 1}, \\
&\qquad p_i\_puts\_down\_f_i, p_i\_puts\_down\_f_{i\oplus 1}, p_i\_gets\_up\} \\
&init_i = all_i = (val(p_i\_has\_f_i, T) \ \underline{or}\ val(p_i\_has\_f_{i\oplus 1}\}, T)) \ \underline{imp}\ val(p_i\_seated, T) \\
&hinv_i = all\_infhists
\end{aligned}
$$

The ith fork $FORK_i$ is a tuple $(C_i, E_i, init_i, all_i, hinv_i)$, where:

$$
\begin{aligned}
&C_i = \{p_i\_has\_f_i, p_{i\ominus 1}\_has\_f_i\} \quad \text{(both of type } Bool\text{)} \\
&E_i = \{p_i\_picks\_up\_f_i, p_{i\ominus 1}\_picks\_up\_f_i, p_i\_puts\_down\_f_i, p_{i\ominus 1}\_puts\_down\_f_i\} \\
&init_i = all_i = not(val(p_i\_has\_f_i, T) \ \underline{\&}\ val(p_{i\ominus 1}\_has\_f_i, T)) \\
&hinv_i = all\_infhists
\end{aligned}
$$

The events, which are to some extent shared, can be defined as follows. These particular definitions will work under both a "strong" and a "weak" interpretation — in this case, unlike that of *Example* 3.13, the "weak" and "strong" sets of histories are identical, both for each fork and philosopher individually, and for the overall composite system.

$p_i\_sits\_down = trans(p_i\_seated, F, T)$
$p_i\_picks\_up\_f_i = trans(p_i\_seated, T, T) \mathbin{\underline{\&}} trans(p_{i\ominus 1}\_has\_f_i, F, F) \mathbin{\underline{\&}} trans(p_i\_has\_f_i, F, T)$
$p_i\_picks\_up\_f_{i\oplus 1} = trans(p_i\_has\_f_i, T, T) \mathbin{\underline{\&}} trans(p_{i\oplus 1}\_has\_f_{i\oplus 1}, F, F) \mathbin{\underline{\&}} trans(p_i\_has\_f_{i\oplus 1}, F, T)$
$p_i\_puts\_down\_f_i = trans(p_i\_has\_f_{i\oplus 1}, T, T) \mathbin{\underline{\&}} trans(p_{i\ominus 1}\_has\_f_i, F, F) \mathbin{\underline{\&}} trans(p_i\_has\_f\_i, T, F)$
$p_i\_puts\_down\_f_{i\oplus 1} = trans(p_i\_has\_f_i, F, F) \mathbin{\underline{\&}} trans(p_{i\oplus 1}\_has\_f_{i\oplus 1}, F, F) \mathbin{\underline{\&}} trans(p_i\_has\_f\_i \oplus 1, T, F)$
$p_i\_gets\_up = trans(p_i\_has\_f_i, F, F) \mathbin{\underline{\&}} trans(p_i\_has\_f_{i\oplus 1}, F, F) \mathbin{\underline{\&}} trans(p_i\_seated, T, F)$

Different philosophers and forks can execute events simultaneously, as long as there is no direct clash, e.g. two different philosophers cannot pick up the same fork at the same time. Also, a fork cannot be simultaneously picked up by one philosopher as it is put down by another. Despite the fairly loose way in which events are defined, state components do not change unless they have to.

The overall system, called $COLLEGE$, is composed from subsystems of philosophers and forks.

$PHILOS = PHIL_0 \mathbin{\underline{\|}} PHIL_1 \mathbin{\underline{\|}} PHIL_2 \mathbin{\underline{\|}} PHIL_3 \mathbin{\underline{\|}} PHIL_4$
$FORKS = FORK_0 \mathbin{\underline{\|}} FORK_1 \mathbin{\underline{\|}} FORK_2 \mathbin{\underline{\|}} FORK_3 \mathbin{\underline{\|}} FORK_4$
$COLLEGE = PHILOS \mathbin{\underline{\|}} FORKS$

As specified, the system can enter deadlock, if all philosophers sit down at the same time, and pick up their own forks at the same time. The system can be modified in various ways to prevent this, e.g. by adding a "butler" system (see [Hoa85]).

The notation used in this example is not particularly perspicuous, but it should be remembered that SSE is not a formalism, but rather a semantic model, being described here in a general meta-language: the objective is simply to show that SSE can model the system by composition, without event promotion. To use SSE in a more convenient way, it would be best to develop a specific formalism, for which SSE would provide the semantics. Such a formalism might combine a Z-like notation for defining states, events, and state invariants, and notations from temporal logic or process algebras for expressing history invariants, along with some new notational features for expressing simultaneity.

4.2. Modelling Events with Inputs and Outputs

Some conventional state-based models — unlike those described in section 2 — use events that correspond not to sets of prestate, poststate pairs, but to n-tuples involving inputs and/or outputs as well as a prestate and a poststate. For example, Z specifications often include events which "query" some aspect of a system state without changing it, returning an output which provides information about the current state. In such an approach, events are not semantically homogeneous, but this presents no problems when concurrency is not allowed. However, in SSE, it is much simpler to represent all events in the same way, so that simultaneous conjunctions and disjunctions of any events can be defined straightforwardly in terms of set intersection and union.

Events which might conventionally have been represented using inputs and outputs can be represented alternatively by regarding certain state components as representing input/output "channels" (of which there may be several, operating in parallel), rather than "internal" system states. A state-querying event could then be represented as one which changes such a channel component, whilst leaving internal state components unchanged. In a specification formalism based on SSE, the intended interpretations of various state components could be distinguished syntactically, e.g. by using "?" and "!" suffixes respectively for state components representing input and output channels. Thus event definitions involving "inputs" and "outputs" would look very similar to those found in conventional formalisms such as Z, although the underlying semantic interpretation of such definitions would be different.

4.3. State Component Granularity

In formalising a system such as the number store of section 1, the choice of state components and their types is somewhat arbitrary. Effectively, Pitt and Byers choose to model it in terms of a single, set-valued state component — although since the component itself is somewhat redundant in a one-component state of the form $\{comp \mapsto val\}$, they just use its value val, which is required to be a subset of $\{1, 2, 3\}$. By comparison, the SSE model in section 3 uses three Boolean-valued state components, $in1$, $in2$, $in3$. Intuitively, these are just different perspectives on the same thing, but the "granularity" of the state components is different. In a single-component model, all state transitions are equivalent in terms of the number of state components that change, because there is only one component to change — whereas in a multiple-component model, some state transitions involve fewer changes to state components than others. In the multiple-component model, the number of state components that change corresponds exactly to the number of changes to the membership of the store, because the membership status of each number is represented explicitly by its own Boolean-valued state component. Thus the multiple-component model is "finer-grained".

The Pitt–Byers semantics is neutral between models with different state component granularities, because it is defined in terms of relations on a state space, without reference to the state components involved in each state. However, this means that it fails to handle the frame problem correctly in *any* model of the number store (or indeed, any model of any other system) that involves disjunctive events of the kind discussed in sections 2 and 3.

By comparison, SSE, as so far presented, is not neutral as regards state component granularity, because although it permits single-component states, the $just_W$ and $just_S$ operators handle the frame problem at the level of multiple state components — given an event definition in terms of certain state components, they constrain what happens to other state components. Consider, however, an attempt to model the number store in SSE, using a single state component in whose possible values are subsets of $\{1, 2, 3\}$, and a loose definition of $add1$ which says that 1 belongs to the value of in in the poststate, but not in the prestate. The problem is that numerous poststate values of in — most involving changes to the membership status of 2 or 3 as well — are sufficient to ensure that this event has occurred, from a prestate in which it is enabled. For example, in the prestate $\{in \mapsto \{2, 3\}\}$, the key sufficient post event sets for the event $add1$ (loosely defined in terms of in), would include e.g. $\{postval(in, \{1\})\}$ (implying a simultaneous removal of 2 and 3, as 1 is added to the store). Thus an occurrence of $add1$, even as the sole event in a system with the state invariant $true$, would allow any concurrent additions or removals of 2 and 3. Neither the $just_W$ or the $just_S$ operator would be able to restrict the poststates reachable, because they are too "coarse-grained" to constrain events loosely defined at the level of a single state component. Trying to redefine $add1$ more tightly would not work either, because it would then not be possible for say, $add1$ and $add2$ to occur simultaneously, assuming a similar tight definition of $add2$.

One response to this issue is just to accept it as a feature of SSE, and to use it to constrain the choice of model when a system is formalised. However, there is another possible approach, which is to introduce the idea of "implicit" state components. For every explicit set-typed state component $sc : \mathbb{P}\, X$, there would be a set of implicit state components $sc_x : Bool$, for each x in X; for every explicit function-typed component $fc : X \to Y$, there would a set of implicit components $fc_x : Y$, for each x in X; and for any explicit tuple-typed component $tc : X_1 \times \ldots \times X_n$, there would be implicit components $tc_1 : X_1, \ldots, tc_n : X_n$. Systems would then be modelled in terms of "full" states, i.e. states that include the implicit components of all their explicit components, and that satisfy the obvious invariants relating each explicit component to its implicit derived components, e.g. if a set-typed explicit component sc had the value val, its Boolean derivative components sc_x would have the value T iff $x \in val$, and F otherwise. (In some cases, this would lead to a system with an infinite set of state components, and possibly with an infinite set of events defined in terms of those components — which is why, in the definition of what a system is, given earlier in section 3.7, the sets C and E were allowed to be infinite.) For example, a full state of the number store with a single explicit state component in would also involve implicit state components $in_1, in_2, in_3 : Bool$, related to in by the appropriate invariants. A model augmented with such extra state components would again be sufficiently "fine-grained" for the $just_W$ and $just_S$ operators to handle loose definitions of $add1$, $add2$, etc. using the corresponding implicit components in_1, in_2, etc. For example, $add1$ would be defined by saying that in_1 changes from F in the prestate to T in the poststate. The development of such ideas is an interesting area for further work.

5. Conclusion

This paper has examined concurrency and the frame problem in state-based formal specification. Simple system state-based models — based on events (sets of prestate, poststate pairs) which are mutually exclusive and deterministic — solve the frame problem by not allowing concurrency, and by ensuring that only one poststate is possible if a given event occurs in a given prestate. However, such events are "overspecified" and inflexible. Pitt and Byers have proposed an alternative semantics, allowing more loosely specified, non-deterministic, and simultaneous events. However, this still constrains events by defining them in terms of particular sets of global states, and its handling of the frame problem is counter-intuitive when applied to certain disjunctive events.

The semantics of simultaneous events (SSE) defines events very loosely, independently of any specific context, frame, or invariant, in a way which makes them reusable without promotion in different systems. Simultaneous events are allowed, and the $just_W$ and $just_S$ operators provide a general mechanism for handling the frame problem.

Future work based on SSE could explore several aspects. One would be to develop a specification formalism based on the SSE semantic model. Another would be to develop software tools for constructing SSE models, and to attempt the formalisation of some more complex systems within SSE. On the theoretical side, it would be interesting to develop related state-based semantic models, which use a similar approach to the frame problem, but represent non-instantaneous events, and various forms of timing and duration constraints. Finally, there seems to be some potential for SSE to provide a semantics for a formal approach to object-orientation with simultaneous concurrency, in which what are currently called "systems" would be equated with object instances.

References

[Bic95] Bicarregui , J. C.: Intra-Modular Structuring in Model-Oriented Specification: Expressing Non-interference with Read and Write Frames. Ph.D. thesis, Dept. of Computer Science, Manchester University, 1995.

[BMR95] A. Borgida, J. Mylopoulos and R. Reiter: On the Frame Problem in Procedure Specifications. *IEEE Trans. on Software Engineering* (1995) 21(10): 785–798.

[CaC96] Cau, A. and Collette, P.: Parallel Composition of Assumption-Commitment Specifications. *Acta Informatica* (1996) 33(2): 153–176.

[Col97] Collette, P.: A Foundation for Modular Reasoning about Safety and Progress Properties of State-Based Concurrent Programs. *Theoretical Computer Science* (1997) 183(2): 253–279.

[Gut85] Guttag, J., Horning, J. J. and Wing, J. W.: Larch in Five Easy Pieces. TR5, DEC Systems Research Center, 1985.

[Hoa85] Hoare, C. A. R.: *Communicating Sequential Processes*. Prentice Hall International, 1985.

[Jon90] Jones, C. B.: *Systematic Software Development Using VDM*. 2nd Edition, Prentice Hall International, 1990.

[Jon91] Jonkers, H. B. M.: Updating the Pre- and Postcondition Technique. In *VDM'91 Formal Software Development Methods*, Lecture Notes in Computer Science, Springer Verlag, 428–456, 1991.

[LaH94] Lano, K. and Haughton, H.: *Object-Oriented Specification Case Studies*. Prentice Hall, 1994.

[Lin96] Lin, F.: Embracing Causality in Specifying the Indeterminate Effects of Actions. In *Proceedings of AAAI'96*, Portland, Oregon, U.S.A., American Association for Artificial Intelligence, 1996.

[MaP92] Manna, Z. and Pnueli, A.: *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1992.

[McH69] McCarthy, J. and Hayes, P. J.: Some Philosophical Problems from the Standpoint of Artificial Intelligence. In D. Michie and B. Meltzer (eds.), *Machine Intelligence 4*, Edinburgh University Press, 463–502, 1969.

[NAT97] The NATURE Team: Defining Visions in Context: Models, Processes and Tools for Requirements Engineering. *Inf. Systems*, Vol.21, No.6, 515–547, 1997.

[PiB94] Pitt, D. H. and Byers, P. J.: The Rest Stays Unchanged (Concurrency and State-Based Specification). *Formal Aspects of Computing* (1994) 6: 471–494.

[PST96] Potter, B., Sinclair, J. and Till, D.: *An Introduction to Formal Specification and Z*. 2nd Edition, Prentice Hall International, 1996.

[Rei91] Reiter, R.: The Frame Problem in the Situation Calculus: A Simple Solution (Sometimes) and a Completeness Result for Goal Regression. In V. Lifschitz (ed.), *Artificial Intelligence and Mathematical Computation: Papers in Honor of John McCarthy*, 359–380. Academic Press, San Diego, CA, 1991.

[Rei96] Reiter, R.: Natural Actions, Concurrency and Continuous Time in the Situation Calculus. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Fifth International Conference (KR'96)*, Cambridge, Massachusetts, U.S.A. November 5-8, 1996.

[ScP86] Schuman, S. A. and Pitt, D. H.: Object Oriented Subsystem Specification. In L. G. Meertens (ed.), *Program Specification and Transformation*. North Holland, 1986.

[SPB90] Schuman, S. A., Pitt, D. H. and Byers, P. J.: Object Oriented Process Specification. University of Surrey, Computing Sciences, Technical Report CS-90-01, 1990.

[Smi00] G. Smith, G.: *The Object-Z Specification Language*. Kluwer Academic Publishers, 2000.

[Spi88] Spivey, J. M.: *Understanding Z: A Specification Language and its Formal Semantics*. Cambridge University Press, 1988.