On Event-B and Control Flow

A. Iliasov

Centre for Software Reliability, Newcastle University, UK
alexei.iliasov@newcastle.ac.uk

Abstract. Event-B is a general purpose formal development method suitable for the design and detailed development of safety-critical systems. Being a data-driven formalism, it lacks any control flow constructs. This turns out to be a limitation for systems with rich control flow properties. In Event-B, control flow information has to be embedded into guards and event actions and this results in an entanglement of control flow and functional specification with the additional downside of extra model variables. This paper proposes a method for extending Event-B models with an new viewpoint portraying control flow properties of a model. The novelty of the work is in relying solely on theorem proving to demonstrate the consistency of control flow and main Event-B specification. The focus is placed on the practicality of working with such an extension and also on achieving proof economy. A detailed formal treatment of the method is presented and illustrated with a case study. A proof of concept implementation for the RODIN platform is briefly discussed.

1 Introduction

Event-B [1–3] is a general-purpose specification language and is a close relative of the popular B-Method [4](or Classical B). Its distinctive feature is the event-based specification paradigm. A model is a collection of a number of events where the next event is selected non-deterministically among the currently enabled events. Event-B facilitates construction of models with a large number of rather simple events. Theorem proving is the primary verification technique and, crucially, almost all the correctness conditions (proof obligations) are formulated on per-event basis. This makes Event-B very friendly to automated theorem provers. High rate of verification automation is extremely important and it makes Event-B one of the few practical proof-based formalisms.

There are some downsides in following pure event-based paradigm. Not all systems are naturally expressed in this style. Often the information about event ordering has to be embedded into guards and event actions. This results in an entanglement of control flow and functional specification with an additional downside of extra model variables.

There are a number of reasons to consider an extension of Event-B with an event ordering mechanism:

- for some problems the information about event ordering is an essential part of requirements; it comes as a natural expectation to be able to adequately reproduce these in a model;
- explicit control flow may help to prove properties related to event ordering;
- sequential code generation requires some form of control flow information;
- since event ordering could restrict the non-determinism in event selection, model checking is likely to be more efficient for a composition of a machine with event ordering information;
- there is a potential for a machine editor presenting a visual machine layout based on control flow information;
- realising such a mechanism could be an initial step towards bridging the gap between high-level workflow and architectural languages and Event-B.

In this paper we discuss an extension of Event-B with a mechanism to reason about event ordering. The practical issues, like verification means, the integration with the Event-B development process and the tooling support are given the highest priority. Unlike much of the work on combining state-based and process-bases specification methods [5–8] our proposal is based on theorem proving rather than model checking. We demonstrate that the proposal is realistic and presents distinct practical advantages with a proof-of-concept tool realising the technique.

2 Background

Event-B is a state-based formalism closely related to Classical B [4] and Action Systems [9]. The step-wise refinement approach is the corner stone of the Event-B development method. The combination of model elaboration, atomicity refinement and data refinement helps to formally transition from high-level architectural models to very detailed, executable specifications ready for code generation. An extensive tool support makes Event-B especially attractive. An integrated Eclipse-based development environment is actively developed and well-supported. Importantly for us, it is open for contributions in the form of Eclipse plug-ins. The main verification technique is theorem proving and the development comes with a collection of powerful theorem provers while there is also a capable model checker.

Formally, an Event-B model is defined by a tuple (c, s, P, v, I, R_I, E) where c and s are constants and sets known in the model; v is a vector of model variables; $P(c, s)$ is a collection of axioms constraining c and s. I is a model invariant limiting the possible states of v: $I(c, s, v)$. The combination of P and I should characterise a non-empty collection of suitable constants, sets and model states: $\exists c, s, v \cdot P(c, s) \wedge I(c, s, v)$. The purpose of an invariant is to express model safety properties (that is, unsafe states may not be reached). In Event-B an invariant is also used to deduce model variable types. R_I is an initialisation action computing initial values for the model variables; it is typically given in the form of a predicate constraining next values of model variables without, however, referring to previous values - $R_I(c, s, v')$. Finally, E is a set of model *events*.

An event is a guarded command:

$$H(c, s, v) \to S(c, s, v, v')$$

where $H(c, s, v)$ is an event guard and $S(c, s, v, v')$ is a before-after predicate. The general form of an event in Event-B notation is

$$\mathit{name} = \texttt{any}\ p\ \texttt{where}\ H(c, s, p, v)\ \texttt{then}\ S(c, s, p, v, v')\ \texttt{end}$$

where p is an a vector of event parameters.

An event may fire as soon as the condition of its guard is satisfied and no other event executes at the same time. In case there is more than one enabled event at a certain state, the demonic choice semantics is applies. The result of an event execution is some new model state v'. The semantics of an Event-B model is usually given in the form of proof semantics, based on Dijkstra's work on weakest precondition. A collection of proof obligations is generated from the definition of the model and these must be discharged in order to demonstrate that the model is correct. For a n abstract model (a model that is not a refinement of another model) two such proof obligations are the invariant satisfaction and event feasibility. A new state produced by an event must satisfy module invariant:

$$I(c, s, v) \land P(c, s) \land H(c, s, v) \land S(c, s, v, v') \Rightarrow I(c, s, v')$$

An event must also be feasible, in a sense that an appropriate new state v' must exist for some given current state v:

$$I(c, s, v) \land P(c, s) \land H(c, s, v) \Rightarrow \exists v' \cdot S(c, s, v, v')$$

There are also proof obligations to establish deadlock freeness, enabledness conditions and a collection of proof obligations for demonstrating Event-B forward simulation refinement [3].

3 Flow Model

The essense of the proposal is an extension of Event-B models with expressions defining event ordering. Such expressions, called *flows*, are written in a special language resembling those used in process algebras, such CSP [10]. The basic element of the language is an event. Events in a flow are the same events as in an Event-B machine. Events are charactersied by an event label and may have parameters (in flow analysis these are treated as an integral part of an event label). The following is the summary of the constructs forming the flow language:

$e.a$	event with label e, index i and arguments a
$p; q$	sequential composition
$p \| q$	parallel composition synchronised on E
$p \sqcap q$	choice
$*(p)$	terminating loop
$'start$	initialisation event
$'stop$	termination event
$'skip$	stuttering event

Events starting with $'$ bear special meaning. $'start$ is a shortcut for Event-B event INITIALISATION, $'stop$ is an assumed termination event and $'skip$ coresponds to an implicit Event-B *skip* event.

An essential part of the flow mechanism is the notion of a *partial flow* expression (or simply partial flow). There are situations when it is not neccessary to mention all the machine events in a flow. For example, one may want to state flow for a part model corresponding to the current refinement step or simply focus on a part where flow reasoning is required. The notion of partial flow becomes clear if one thinks of a flow expression as a set of conditions formulated on a machine. A partial flow is then a more relaxed version of a complete flow.

There are some basic well-formedness requierements to a flow. Event $'start$ corresponding to the initalisation event of a machine may not be composed with other events using choice and parallel composition. Also, it may only occur on the left-hand side of a sequential composition. This restriction is due to the fact that the initialisation event is a special case in Event-B. It has no guard and is always a first event to run. Since flows may be partial, initialisation event may be omitted from a flow. The termination event $'stop$ also needs special treatment. This event is not present explicitly in a machine and the following Event-B definition is implied if the event is present in a flow expression:

$$'stop = \texttt{when}\ \neg(G_1 \vee \cdots \vee G_n)\ \texttt{then skip end}$$

$'stop$ is enabled when all other events are disabled; it executes infinitely but keeps the state intact so that a machine cannot get into a state when anything else is enabled. Since this event diverges it is not possible to have any other event to follow $'stop$. Hence, $'stop$ may not occur on the right-hand side of a sequential composition. For the same reason, a parallel composition with $'stop$ is disallowed. It is possible, however, to have a choice between $'stop$ and another event (including $'start$).

In a composition of a flow and machine, the flow loop construct $*(p)$ would correspond to a loop on machine events. It is the responsibility of the Event-B part to demonstrate the convergence of a loop. This is a standard part of model analysis in the RODIN Event-B environment. Later we discuss how to improve the strategy of demonstrating convergence in Event-B by using the information contained in a flow attached to a machine.

In the context of Event-B models, the parallel composition may only be applied to certain kind events. We require that for any set of parallel events (as defined in a flow expression) there exists a well-formed Event-B event that simulates all the possible interleavings of the parallel events. This condition results in a number of syntactic requirements to machine events.

Let $rd(e)$ return the set of all variables read by event e. These are the model variables referenced in the event guard and the event actions. Likewise, $wr(e)$ is a set of variables updated by event e. These are the variables found on the left-hand side of substitutions in an event body. Events that are potentially concurrent are called independent events.

Definition 1. Independent events. *Events that do not have read/write and write/write conflict are independent. The conflicts are defined as follows:*

- Read/write conflict. *A pair of events have a read conflict if one updates the variables read by another. This is denoted as* $rdcfl(e_1, e_2) = rd(e_1) \cap wr(e_2) \neq \varnothing \vee rd(e_2) \cap wr(e_1) \neq \varnothing$.
- Write/write conflict. *Events updating the same variable have a write conflict:* $wrcfl(e_1, e_2) = wr(e_1) \cap wr(e_2) \neq \varnothing$.

Formally, set E *of events is independent, denoted as* $ind(E)$*, if for every event pair* (a, b) *from* E *the following holds:* $a \neq b \implies \neg rdcfl(a, b) \wedge \neg wrcfl(a, b)$

The condition $ind(\dots)$ is checked for all the possible pairs of events composed with the parallel composition operator.

4 Semantics

This section discusses the semantics of the flow language and the way to integrate it with Event-B. In particular we show how to reason about flow and machine consistency in the terms of machine properties rather than flow or machine traces. But first we use the traces semantics to formally integrate flows with Event-B. The following defines the traces of a flow expression.

$$
\begin{array}{ll}
traces('skip) & \mathrel{\widehat{=}} \{\langle\rangle\} \\
traces('start) & \mathrel{\widehat{=}} \{\langle 'start\rangle\} \\
traces('stop) & \mathrel{\widehat{=}} \{s \mid n \in \mathbb{N} \wedge s \leq \{\langle 'stop\rangle\}^n\} \\
traces(e_i.a) & \mathrel{\widehat{=}} \{\langle e_i.a\rangle\} \\
traces(p; q) & \mathrel{\widehat{=}} \{s^\frown z \mid s^\frown z \in traces(p) \wedge z = \langle 'stop\rangle\} \\
 & \quad \{s^\frown t \mid s^\frown z \in traces(p) \wedge t \in traces(q) \wedge z \neq \langle 'stop\rangle\} \\
traces(p|q) & \mathrel{\widehat{=}} traces(p) \cup traces(q) \\
traces(*(p)) & \mathrel{\widehat{=}} traces(p|(p;*(p))) \\
traces(p\|q) & \mathrel{\widehat{=}} \{s \overline{\|}_E t \mid s \in traces(p) \wedge t \in traces(q)\}
\end{array}
$$

Here $s \leq t$ states that trace s is a prefix of trace t; $\alpha(x)$ is an alphabet of x (set of all events occurring in x). The parallel composition operator is defined as a collection of all the possible event interleavings:

$$\begin{array}{lll}
\langle\rangle \overline{\|}_E \langle\rangle & \mathrel{\widehat{=}} \langle\rangle & \\
\langle a\rangle^\frown p \overline{\|}_E \langle\rangle & \mathrel{\widehat{=}} \oslash & a \in E \\
\langle a\rangle^\frown p \overline{\|}_E \langle\rangle & \mathrel{\widehat{=}} \{\langle a\rangle^\frown s | s \in (p \overline{\|}_E \langle\rangle)\} & a \notin E \\
\langle a\rangle^\frown p \overline{\|}_E \langle b\rangle^\frown q & \mathrel{\widehat{=}} \oslash & a \in E \wedge b \in E \wedge a \neq b \\
\langle a\rangle^\frown p \overline{\|}_E \langle b\rangle^\frown q & \mathrel{\widehat{=}} \{\langle a\rangle^\frown s | s \in (p \overline{\|}_E q)\} & a \in E \wedge b \in E \wedge a = b \\
\langle a\rangle^\frown p \overline{\|}_E \langle b\rangle^\frown q & \mathrel{\widehat{=}} \{\langle b\rangle^\frown s | s \in (\langle a\rangle^\frown p \overline{\|}_E q)\} & a \in E \wedge b \notin E \\
\langle a\rangle^\frown p \overline{\|}_E \langle b\rangle^\frown q & \mathrel{\widehat{=}} \{\langle a\rangle^\frown s | s \in (p \overline{\|}_E \langle b\rangle^\frown q)\} \cup & a \notin E \wedge b \notin E \\
 & \quad \{\langle b\rangle^\frown s | s \in (\langle a\rangle^\frown p \overline{\|}_E q)\} &
\end{array}$$

$p \overline{\|}_E q$ constructs all the possible interleavings of p and q while respecting the synchronisation on common events E.

4.1 Event-B Trace Semantics

In this section we briefly present how traces of an Event-B model are constructed. Much more detailed treatment of the subject is given in [11] and [12].

An elementary step of a machine interpretation is the computation of the set of next states for some current event. For some event e the next states are found by selecting a set of suitable values for the event parameters and using them to characterise the possible next states v'. An Event-B machine may be understood as a relation $T : \mathit{Event} \leftrightarrow S \leftrightarrow S$:

$$T \overset{\mathrm{df}}{=} \exists p_e \cdot (G_e(p_e, v) \wedge S_e(p_e, v, v'))$$

where p_e, G_e, S_e are the event parameters, guard and before-after predicate. T is a predicate characterising a a relation on system states.

A next event would start from a state produced by a previous event. This is expressed with the sequential composition operator ; defined as follows:

$$e_1; e_2 = T(e_1)[v_1/v'] \wedge T(e_2)[v_1/v]$$

where v_1 is a vector of fresh names used to record the final state of e_1 and pass it on to e_2.

The concept of sequential composition can be generalised to a chain of events. Operator *seq* performs a sequential composition over an event list:

$$\begin{array}{l}
\mathit{seq}(\langle\rangle) = T(\mathit{skip}) \\
\mathit{seq}(\langle e\rangle t) = T(e); \mathit{seq}(t)
\end{array}$$

Using this definition, the traces of a machine are defined as all possible traces reachable from the initial machine state:

$$\mathit{traces}(M) = \{t \mid \mathit{seq}(t)[\mathit{Init}] \neq \oslash\}$$

In the next section we use the traces semantics of flows and Event-B to define the consistency conditions for a model combining a flow expression and an Event-B machine.

4.2 Flow/Machine Consistency

The minimal requirement to a given pair of a flow and machine is that the two agree on deadlocks and divergences. To account for partial flows it is required to consider a situation when only a part of a machine traces is specified by a flow. A flow trace starting with $'start$ and eventually reaching $stop$ would match a complete machine trace if it matches any trace at all.

Definition 2. Flow consistency. *A flow is consistent with a given machine if it is possible to find a machine trace that contains some flow trace:* $\exists t, hd, tl \cdot t \in traces(f) \wedge hd^\frown t^\frown tl \in traces(m)$.

One important case of a flow and machine combination is when flow event ordering and event guards together define a concrete, implementable event ordering. Individually, both flow expression and machine still may have non-deterministic event choice. Such property is essential for code generation and sometimes is a desired property of a model. While choice related non-determinism must be resolved, non-deterministic event ordering may still be present due to the parallel composition operator. To distinguish between these two cases we use the notion of interleaving equivalence.

Two traces are said to be interleave equivalent if one can be obtained from another by swapping events in a pair of independent events. This is states with a help of relation Re defined on traces as follows: $s \; Re \; t \Leftrightarrow s = t \vee \exists a, b, hd, tl \cdot (hd^\frown \langle a, b\rangle^\frown tl \in t \wedge hd^\frown \langle b, a\rangle^\frown tl \in s \wedge ind(\{a, b\}))$

Traces s and t are said to be interleave equivalent if $s \; Re^* \; t$ where Re^* is a transitive closure of Re.

Definition 3. Concrete flow. *The traces contained in the intersection of a concrete flow and machine traces are interleave equivalent.*

Having these definitions does not lead to practical means of establishing flow properties especially since it is our intention is to use theorem proving to reason about a combination of a flow and machine. In the rest of the section we discuss how to transition from statements about traces of flows and machines to equivalent conditions on machine variables, events guards and event actions. First, some mathematical context is presented. This gives a basis for theorems reformulating the definitions of consistent and concrete flows in the terms of machine properties. In its turn, this gives a foundation for deriving proof obligations.

In a general case, an event may be preceded by any configuration of choice and parallel composition. Let us consider the following example: $((a\|b)|(c\|d)); z$. Event z gets enabled as soon as both a and b or c and d terminate. One has to show that for any possible situation (that is, the first or the second branch of the choice) it is possible to pass control to z. Even more complex case is demonstrated by the following expression: $((a\|b)|(c\|d)); ((e\|f)|(g\|h))$. For this, one also has to consider a multitude of options on the right-hand side. The notions of *entry and exit points* are introduced to reason about events actively involved in passing control in a sequential composition. These are defined as follows:

$$
\begin{array}{llll}
EN(e) & = \{\{e\}\} & EX(e) & = \{\{e\}\} \\
EN('skip) & = \{\{\}\} & EX('skip) & = \{\{\}\} \\
EN('start) & = \{\{'start\}\} & EX('start) & = \{\{'start\}\} \\
EN('stop) & = \{\{'stop\}\} & EX('stop) & = \{\{'stop\}\} \\
EN(p;q) & = EN(p) \quad p \neq' skip & EX(p;q) & = EX(q) \quad q \neq' skip \\
EN('skip;q) & = EN(p) & EX(p;' skip) & = EX(p) \\
EN(p|q) & = EN(p) \cup EN(q) & EX(p|q) & = EX(p) \cup EX(q) \\
EN(p\|q) & = \{\{EN(p), EN(q)\}\} & EX(p\|q) & = \{\{EX(p), EX(q)\}\} \\
EN(*(p)) & = EN(p) & EX(*(p)) & = EX(p)
\end{array}
$$

where $EN(x)$ is a set of entry points of a flow expression x. Correspondingly, $EX(x)$ denotes the set of exit points. Note that entry and exits points are set of sets. The reason is that a combination of parallel composition and choice results in a set of event clusters. For example the set of entry points of $((a\|b)|(c\|d)); z$ is $\{\{a, b\}, \{c, d\}\}$. This set contains two entry points $\{a, b\}$ and $\{c, d\}$ where each entry points is set itself denoting a complex entry point of a parallel composition construct.

Independent events may be *merged* into a single event[1]. Indeed, since independent events are conflict free and can be executed in any order there is nothing that prevents an existence of a single event that would have the same effect as all possible interleavings of the independent events. This is a purely abstract construction. There is, of course, no need to actually introduce merged events in a model.

Definition 4. Operator $merge(a, b)$. *The operator constructs a single event from the definitions of events a and b. It is well-defined only when a and b are independent. For some events a and b*

$$
\begin{array}{l}
a = \texttt{any } p \texttt{ where } G(p, d) \texttt{ then } S(p, d, w') \texttt{ end} \\
b = \texttt{any } q \texttt{ where } H(q, g) \texttt{ then } R(q, g, u') \texttt{ end}
\end{array}
$$

A merged event has the following general form:

$$
a = \texttt{any } p, q \texttt{ where } G(p, v) \wedge H(q, v) \texttt{ then } S(p, v, v') \wedge R(q, v, v') \texttt{ end}
$$

Since only independent events may be merged, the resultant merged event enjoys a number of properties. It is enabled when both its donor events are enabled and simulates the effect of interleaving the merged events. A merged event is feasible as long as its individual donor events are feasible. It is straightforward to see that the state observed after executing a merged event is the same state as one would observe after executing both donor events in any order. Event merging is a special case of *event fusion* [13].

[1] Event-B uses event merging as a refinement technique. This has nothing to do with our definition of merging.

Definition 5. Operator $s \fatsemi_m t$. *This operator defines the consistency conditions for a sequential composition where control is passed from a collection of exit points s to a collection of entry points t. The operator type is*

$$\fatsemi : M \times \mathbb{P}(\mathbb{P}(Event)) \times \mathbb{P}(\mathbb{P}(Event)) \to BOOL$$

where M is an Event-B model and Event is a set of model events; the second and the third parameters are some exit and entry points.

The strategy is to construct an Event-B event implementing what is essentially a sequential composition of s and t. The feasibility conditions for the event would demonstrate the well-formedness of a sequential composition.

Let us first consider a simple case of a composition of two events when $s = \{\{e_1\}\}$ and $t = \{\{e_2\}\}$. Events e_1 and e_2 are defined as follows (these definitions come from an Event-B machine that is supplied as the first parameter to operator):

$$e_1 = \texttt{any}\ p\ \texttt{where}\ G(p,v)\ \texttt{then}\ S(p,v,v')\ \texttt{end}$$
$$e_2 = \texttt{any}\ q\ \texttt{where}\ H(q,v)\ \texttt{then}\ R(q,v,v')\ \texttt{end}$$

A composed event "$e_1; e_2$" is an event with the same guard as e_1 and the after state of e_2 when executed after executing e_1:

$$"e_1; e_2" = \texttt{any}\ p\ \texttt{where}\ G(p,v)\ \texttt{then}\ S(p,v,v'); (\exists q \cdot H(q,v) \land R(q,v,v'))\ \texttt{end}$$

Here we introduce operator ; for the sequential composition of event actions[2]*. It can be reduced to a simple action using the following definition:*

$$S_0(p,v,v'); S_1(p,v,v') \mathrel{\widehat{=}} \exists\ v_1\ \cdot\ S_0(p,v,v_1)\ \land\ S_1(p,v_1,v')$$

Now we are ready to define the meaning of $\fatsemi_m$ when, as a special case, it is applied to a pair of events:

$$e_1 \hat{\fatsemi}_m e_2 = \mathsf{FIS}("e_1; e_2")$$

where $\mathsf{FIS}(e)$ is an Event-B event feasibility condition (see Section 2 and also [3]).

The next step is to reduce the general form of $\fatsemi$ to the simple case above. For this we consider all the pairs from a cartesian product of s and t while also reducing the multiple exit and entry points introduced by the parallel composition construct to a single event.

$$\forall e_1, e_2 \cdot (e_1, e_2) \in s \times t \Rightarrow mergeall(e_1) \hat{\fatsemi}_m mergeall(e_2)$$

where $mergeall(x)$ is a following generalisation of merge:

[2] Classical B defines a similar operator to compose actions[4].

$$mergeall(x) = \begin{cases} e & x = \{e\} \\ merge(hd, mergeall(tl)) & x = \{hd\} \cup tl \end{cases}$$

Finally, we are ready to approach the problem of checking flow/machine consistency. Using the $\fatsemi$ operator, the problem is reduced to a number of conditions on Event-B machine events. Importantly, they all are expressed in first-order logic as they are essentially various instance of the Event-B feasibility proof obligation. The last remaining step is to lift $\fatsemi$ to the level of a model composed of a machine and flow.

Definition 6. Predicate *cons. This predicate defines the consistency conditions for a combination of a flow and machine. Its type is* $cons : F \times M \to BOOL$ *and the definition is as follows:*

$$\begin{aligned} cons(ev, m) &= true \\ cons(p; q, m) &= cons(p) \land cons(q) \land (EX(p) \fatsemi_m EN(q)) \\ cons(p|q, m) &= cons(p) \land cons(q) \\ cons(p\|q, m) &= cons(p) \land cons(q) \\ cons(*(p), m) &= cons(p) \end{aligned}$$

where ev is either a machine event one of the predefined events ('skip, 'start or 'stop).

Now we are able to state the flow consistency as a condition on machine elements.

Theorem 1. *A flow f is consistent with a machine m provided $cons(f, m)$ holds.*

Proof. There are two fundamental reasons why a trace required by the Definition 2 could not be found.

Firstly, either a flow or machine may diverge at different points without giving an option to continue with a non-divergent trace. For a flow this could only happen when there is a transition into $'stop$ event (flow loops always agree with machine event loops on divergences since a flow loop covers both terminating and non-terminating machine loops). In other words, there is an instance of sequential composition $p; q$ such that $\{'stop\} \in EN(q)$. For a machine, a divergence on traces happens when an event infinitely enables itself while keeping all other events disabled. The conditions introduced by *cons* guarantee that any sequential composition is consistent and thus a divergent event may not be found in the entry points of the right-hand side of a sequential composition. Then, assuming that flow and machine traces agree on deadlocks, such an event may only be $'stop$. Hence, the satisfaction of $cons(f, m)$ establishes the fact that traces of f and traces of m agree on divergences.

Secondly, there is a possibility that a combination of a flow and machine reveals deadlocks that were not present in either flow or machine alone. The only source of such deadlocks is a sequential composition that is not well-formed. However, $cons(f, m)$ states that this may not be a case.

Flow made of a loop of choices over machine events (including $'stop$) is always consistent with the machine. Indeed, for any given event, possible continuation is either another machine event or a termination.

Corollary 1. *Flow model $*(e_1|\ldots|e_k|'stop)$ is consistent with any (well-formed) Event-B machine with events $e_1, \ldots, e_k$. This immediately follows from the definition of cons as this flow expression does not contain sequential or parallel composition and thus is free from any consistency obligations.*

One interpretation of an Event-B machine is that of a loop made of machine events and preceded by the initialisation event. In the flow language this is expressed as $'start; *(e_1|\ldots|e_k)$. This expression gives rise to a consistency condition requiring that there is an enabled event after the initialisation event. It is straightforward to see that machines shown to be deadlock free or refining a deadlock free abstract machine are always consistent with this flow.

Theorem 2. *A consistent flow f, containing $'start$ in its traces, is concrete with the respect to machine m if for every instance of the sequential composition $p; q$ the following condition holds: $\forall s, t \cdot \{s, t\} \in EN(q) \land s \neq t \implies \neg(EX(p) \mathbin{⨾}_m s \land EX(p) \mathbin{⨾}_m t)$*

Proof. Let us consider two traces of f: d and g, $d \in traces(f), g \in traces(f)$ such that they are prefixes of some machine traces: $\exists md, mg \cdot d \leq md \land g \leq mg$. d and g are necessarily prefixes since $'start$ is included in the flow expression f. Should it not be possible to find two machine traces then the theorem condition is trivially satisfied. Let us assume that d and g are not interleave equivalent: $\neg(d\ Re^*\ g)$. Then it is possible to find two distinct, non-independent events a and b, $a \neq b, \neg ind(a, b)$ where $\exists hd \cdot hd ^\frown \langle a \rangle \leq d \land hd ^\frown \langle b \rangle \leq g \land \#hd > 0$ and $\#x$ denotes the length of trace x. The two traces record the same event occurrences until a point when a is recorded in one and b is recorded in another. Since the theorem condition requires that f uses $'start$ it is known that $\langle 'start \rangle \leq hd$ and thus hd is not empty. Prefix hd corresponds to some flow expression fp such that $traces(fp) = hd$ (it is not, however, necessarily a part of f as it might be just one possible trace of a parallel composition in f). The fact that d and g disagree on events a and b necessarily requires that pf is followed by a choice construct that among its entry points has a and b. Thus, machine definition would have to satisfy the following condition: $EX(fp) \mathbin{⨾}_m \{a\} \land EX(fp) \mathbin{⨾}_m \{b\}$. Let us consider the theorem condition where let $p = fp$ and $\{a, b\} \in EX(q)$. Then $\neg(EX(fp) \mathbin{⨾}_m \{a\} \land EX(fp) \mathbin{⨾}_m \{b\})$. The contradiction proves the theorem.

These two theorems show how to reason about flow and machine consistency in terms of conditions o machine elements. Next we show how derive conditions that could be used as proof obligations in the automated reasoning framework of RODIN Platform[14].

4.3 Proof Obligations

To build a tool realising a language extension such ours one would need to ensure that there are suitable verification means. Event-B is designed specifically for theorem proving. It facilitates the creation of a large number of simple proof obligations by requiring a modeller to express complex state transitions with a number of simple atomic steps. The absolute number of proof obligations is not of a major concern since there is an automated theorem proving support able to discharge the majority of them. Still proofs scalability is important as to not overwhelm provers with a gigantic number of proof obligations. In Event-B, the number of proof obligations is approximately in linear correspondence with the number of major model elements: events, invariants, guards and actions.

For a combination of a flow and machine we would like to be able to demonstrate that the flow is consistent or concrete (the latter requires the former). The general strategy is split an overall proof into a collection of simpler conditions.

For flow consistency, a suitable way to do this is to analyse each instance of sequential composition individually as suggested by the condition of Theorem 1 (see Definition 6 for operator *cons*). For an instance of a sequential composition, from Definition 5 we have the following feasibility condition for a composed event.

$$\begin{array}{l} I(v) \wedge G(p,v) \vdash \\ \quad \exists v' \cdot (S(p,v,v'); (\exists q \cdot H(q,v) \wedge R(q,v,v'))) \vdash \\ \qquad \exists\ v_1\ \cdot\ (S(p,v,v_1)\ \wedge\ \exists q \cdot H(q,v_1) \wedge R(q,v_1,v'))) \end{array}$$

The condition is far too complex in the current form. A more compact one could be found. Let us first assume that the composed events are feasible on their own. This gives the following two axioms.

$$\begin{array}{l} \mathrm{axm1} : I(v) \wedge G(p,v) \vdash \exists v' \cdot S(p,v,v') \\ \mathrm{axm2} : I(v) \wedge H(q,v) \vdash \exists v' \cdot R(q,v,v') \end{array}$$

Applying axiom axm1, the feasibility condition for a composed event is simplified to the following:

$$I(v) \wedge G(p,v) \wedge S(p,v,v_1) \vdash \exists\ q\ \cdot\ H(q,v_1) \wedge R(q,v_1,v')$$

With the help of the second axiom we are able to remove $R(q,v_1,v')$ clause from the goal:

$$I(v) \wedge G(p,v) \wedge S(p,v,v_1) \vdash \exists\ q \cdot\ H(q,v_1)$$

Finally, extending the above with the consideration of model constants and sets, the following proof obligation is formulated.

$$P(c,s) \wedge I(c,s,v) \wedge G(c,s,p_e,v) \wedge S(c,s,p_e,v,v') \vdash H(c,s,q,v) \qquad (1)$$

Here G and S are the guard and before-after predicate (actions) of what is possibly a result of merging several model events. The proof obligation demonstrates that an event characterised by G and S is able to pass control to another (possibly merged) event with guard H for any possible state permitted by G.

The axioms we have relied upon are sound since they are a part of model consistency proof obligations that are to be discharge for every Event-B model[3].

With a similar procedure we able to find a practical form of a proof obligation for demonstrating that a flow is concrete. The following proof obligation requires that for a given instance $p; q$ of a sequential composition the choice branches in q, if there any, are mutually exclusive.

$$\begin{array}{c} P(c,s) \wedge I(c,s,v) \wedge G(c,s,p,v) \wedge S(c,s,p,v,v') \vdash \\ \bigwedge_{\{s,t\} \in EN(q) \wedge s \neq t} \neg(H_s(c,s,q_s,v') \wedge H_t(c,s,q_t,v')) \end{array} \tag{2}$$

Here H_s and H_t are the guards of possibly merged events. The goal in this proof obligation may become lengthy in some extreme case when there is a choice on a large number of events. However, since the goal is in conjunctive form is relatively straightforward for a prover to apply case analysis.

4.4 Example

In this section we consider a combination of a simple Event-B model and flow expression. An emphasis is made on using sequential event composition as it is the construct requiring the consistency proof obligations.

The example is a sluice with two doors connecting areas with dramatically different pressures. The pressure difference makes it unsafe to open a door unless the pressure is levelled between the areas connected by the door. The purpose of the system is to adjust the pressure in the sluice area and control the door locks to allow a user to get safely through the sluice.

The model has three variables: $d1 \in DR$ and $d2 \in DR$ are the door states; $pr \in PR$ is the current pressure in the sluice area. A door is either closed or open: $DR = \{OP, CL\}$ and pressure is low or high: $PR = \{HIGH, LOW\}$. Initially, the doors are shut and the pressure is set to low.

A model has a number of invariants expressing the safety properties of the system: a door may be opened only if the pressures in the locations it connects is equalised; at most one door is open at any moment; the pressure can only be switched on when the doors are closed. Model events control the doors and a device regulating the sluice pressure:

$$\begin{array}{ll}
open1 & = \texttt{when}\ d1 = CL \wedge pr = LOW\ \texttt{then}\ d1 := OP\ \texttt{end} \\
close1 & = \texttt{when}\ d1 = OP\ \texttt{then}\ d1 := CL\ \texttt{end} \\
open2 & = \texttt{when}\ d2 = CL \wedge pr = HIGH\ \texttt{then}\ d2 := OP\ \texttt{end} \\
close2 & = \texttt{when}\ d2 = OP\ \texttt{then}\ d2 := CL\ \texttt{end} \\
pr\_low & = \texttt{when}\ d1 = CL \wedge d2 = CL \wedge pr = HIGH\ \texttt{then}\ pr := LOW\ \texttt{end} \\
pr\_high & = \texttt{when}\ d1 = CL \wedge d2 = CL \wedge pr = LOW\ \texttt{then}\ pr := HIGH\ \texttt{end}
\end{array}$$

Finally, the following flow expression is aused. It describes a sequence of steps needed to let a user through the sluice starting from an area adjoining door 1 ($d1$): $pr\_low; open1; close1; pr\_high; open2; close2$

Let us see how we can check that this specification is consistent with the flow expression. For each instance of sequential composition (*pr_low*; *open*1, *open*1; *close*1 and so on) it is needed to show that condition (1) holds. For example, for *pr_low*; *open*1 it is:

$$\begin{cases} d1 = CL \wedge d2 = CL \\ pr' = LOW \wedge d1' = d1 \wedge d2' = d2 \end{cases} \vdash d1' = CL \wedge pr' = LOW$$

The condition is trivially true. Another proof obligation, generated *open*1; *close*1, also trivially holds:

$$\begin{cases} d1 = CL \wedge pr = LOW \\ pr' = pr \wedge d1' = OP \wedge d2' = d2 \end{cases} \vdash d1' = OP$$

The next case presents some difficulties. When trying to demonstrate that event *close*1 always enables *pr_high* we find that there is not enough information to discharge the proof obligation:

$$\begin{cases} d1 = OP \wedge pr' = pr \\ d1' = CL \wedge d2' = d2 \end{cases} \vdash d1' = CL \wedge d2' = CL \wedge pr' = LOW$$

The problem here is that event *close*1 is more general than it needs to be. By strengthening its guard with the additional clauses $d2 = CL \wedge pr = LOW$ we are able to discharge the proof obligation. However, the guard strengthening invalidates other flow proof obligations and requires changes to other events. This is still not hard to do and the result is a model where all the proof obligations are discharged automatically.

4.5 Collecting Additional Hypothesis

There is a way to discharge proof obligations like this without strengthening event guards. Indeed, by looking at the flow expression one should notice that *close*1 is always preceded by *pr_low* and thus may only be enabled when $pr = LOW$. Likewise, since *close*1 always follows *open*1 and the second door is always closed in the after-states of *open*1 (due to the safety invariant of the model requiring that at most one door is open a time) it is known that the condition $d2 = CL$ is always true for states when *close*1 is enabled. Hence all the information that was introduced into proofs by strengthening event guards is already present in a model. To benefit from this information it must be collected and added in the form of hypothesis to flow proof obligations.

Let v_{i-1} be a model state preceding state v_i and state v_n be the most recent previous state preceding the current state v. Also, let $H_i(v_1, \ldots, v_n, v)$ be the current collection of hypothesis for some event a. Then for an instance of sequential composition $a; b$ the collection of hypothesis available in the after-state of b is computed as

$$H_{i+1} = H_i(v_1, \ldots, v_n, v_{n+1}) \wedge G(v_{n+1}) \wedge S(v_{n+1}, v)$$

where G and S are the guard and actions of b. It is straightforward to generalise this basic procedure to the complete flow language. However, there an issue of filtering out irrelevant hypothesis as a large number of hypothesis slows down some provers.

4.6 Flow Refinement

Event-B developments are constructed in a gradual manner starting with an abstract model and arriving at a runnable specification through a number of refinement steps. Since flow is a part of a model there must be a way to state a refinement relation between the flow of an abstract model and the flow of a concrete model. We use the traces refinement to define ths refinement relation. Th traces refinement is easy to check with a tool and it turns out, in our case, to be equivalent to demonstrating a much stronger failure-divergence refinement condition. To keep flow events in agreement with machine events, some renaming is applied before comparing flow traces:

$$f_a \sqsubseteq f_c \Leftrightarrow traces(R^*(f_c \setminus E_n)) \subseteq traces(f_a)$$

where $x \setminus S$ removes all occurrences of events from S in traces of x; E_n is a set of new events introduced in machine refinement (these events refine an implicit *skip* event of an abstract machine); R^* is a function mapping concrete event labels into the labels of abstract events.

For flows of abstract and concrete models considered in isolation the refinement relation the traces refinement. However, once the consistency of a flow and machine is taken into account and machine refinement obligations are satisfied, a concrete model made of a flow and Event-B machine respects the failure-divergence refinement. This is due to the fact that Event-B refinement formulated in terms of traces is itself a case failure-divergence refinement [12].

4.7 Reasoning about Flows

With the addition of a flow, it is possible to reason to a certain degree about deadlock freeness, liveness and reachability properties of a model. The Event-B approach to deadlock freeness is to demonstrate that events guards exhaust the model invariant. In other words, for any state permitted by invariant there is an enabled event: $I(v) \implies G_1(v) \vee \cdots \vee G_n(v)$

The condition turns out to be too strong for larger models as it requires formulating the strongest possible invariant and this is not always practical. On the other hand, a flow attached to a model leads to a number of proof obligations which satisfaction demonstrates the unfailing progress of a model through the events of a flow expression. The only way to introduce termination in a complete flow (a flow covering all the events of a machine) is to explicitly use event $'stop$. This makes reasoning about deadlocking and termination more intuitive.

A flow expression may be seen as a directed graph. Its vertices are model events and edges are the transitions connecting events in a flow expression.

property	*definition*	*description*
eventually	$a\ F^*\ b$	after a eventually b
reachable	$'start\ F^*\ b$	b is reachable
always reachable	$\forall e \cdot' start\ F^*\ e \Rightarrow e\ F^*\ b$	b is always reachable
liveness	$\forall e \cdot' start\ F^*\ e \Rightarrow \exists n \cdot \{b\} = F^n(e)$	b keeps happening

Fig. 1. Flow properties

Computing the transitive closure of such graph, it is possible to check statements like "*after event a eventually event b*" or "*event x is reachable*". Let us assume that F is a graph constructed from a flow expression: $F : Event \leftrightarrow Event$. Then "*after event a eventually event b*" may be stated as $a\ F^*\ b$ and "*event x is reachable*" as $'start\ F^*\ x$. One can also check that event x is always reachable by stating that it can be reached from any event that in its turn is reachable from the initialisation event: $\forall e \cdot e \in F^*('start) \Rightarrow e\ F^*\ x$. One can also express liveness properties to check that something good keeps happening throughout a system lifetime (Figure 1).

It is important to understand how to interpret such properties. Since they are checked at the level of a flow and a flow may have more traces than a machine, not all flow properties automatically hold for a combination of a flow and machine. The source of this unfortunate complication is the choice construct: the composition with machine may eliminate some choice branches and make events reachable in a flow unreachable in a composite system. In such situations a help from a model checker [15] or an animator should be procured to ascertain that each flow branch is reachable. It this light, formulating flow properties may seem a vain exercise. However, flows give a considerable advantage in model checking by reducing a model state space. Since validating flow properties is computationally cheap and user gets an instant feedback, it is better to first constrain a flow expression as much as possible and then apply more general model checking techniques.

4.8 Convergence

One of the proof obligations of the Event-B refinement relation is the demonstration of the fact that all new events (machine events not refining abstract events) *converge* in the sense they may not stay indefinitely enabled without passing control to an event refining an abstract event. The technique is to find a well-founded expression, called variant, decremented by each new event. Since such variant must be common for all the new events it may be quite difficult to find a suitable expression. It is not unusual to introduce auxiliary variables to help with expressing a variant. It is even more difficult to deal with cases when, conceptually, there are two or more loops on new events.

Since flow has an explicit notion of a loop it brings some flexibility into demonstrating convergence of new events. For instance, if an event is not a part of a loop in a flow then there is no need to demonstrate its convergence: the flow proof obligations guarantee that it passes control to some other event (in this discussion we assume that flows are not partial and cover all the model events).

The following procedure checks the convergence of a model using the information from a flow expression. The second parameter of *conv* is a variant expression - in addition to having machine-level variants we propose to attach variants to flow loops (in $*(p) : w$, w is a variant associated with loop $*(p)$):

$$\begin{array}{lll}
conv(p; q, v) & = conv(p) \vee conv(q) & \\
conv(p|q, v) & = conv(p) \wedge conv(q) & \\
conv(p\|q, v) & = conv(p) \wedge conv(q) & \\
conv(*(p) : w, v) & = conv(p, w) & \\
conv(e, v) & = true & e \in \{'skip,' start,' stop\} \vee inherited(e) \\
conv(e, v) & = \mathrm{VAR}(e, v) & new(e)
\end{array}$$

Here $inherited(e)$ states that event e refines an abstract event; $new(e)$ requires that e is a new event; $\mathrm{VAR}(e, v)$ is the Event-b convergence proof obligation [3] for event e with variant expression v. One can have a significant proof economy by demonstrating convergence in this way. For example, there are no convergence proof obligations for a flow $*(a; b)$ where a is an inherited event. Without a flow, one would still have to prove that b converges.

5 Tool Support

The proposed mechanism has been implemented as an extension of RODIN platform [14]. The platform is an Eclipse-based integrated environment for constructing Event-B developments. It provides means for model manipulation (editing, pretty-printing, exporting, etc.) and verification. The platform is responsible for generating proof obligations demonstrating model consistency and also the refinement obligations if a model happens to be a refinement of another model. Proof obligations are handed over to a collection of theorem provers. Any unproved obligations has to be analysed in an integrated interactive prover.

We considered it essential to make the flow extension a natural part of an Event-B development method. The flow editing is done in a new section of a structured model editor. Flow proof obligations are automatically generated from a flow expression attached to a machine. Syntactic checks and flow refinement checks are also done automatically in a background while a user works with a model. RODIN extensions, including our flow tool, are realised as Eclipse plug-ins. There is a considerable number of plug-ins available for the platform including model checking [15], animation, requirements analysis, UML/Event-B integration and others [14].

A number of case studies were carried out using the tool. One of the large ones is the development of the sluice control system related to the example given earlier in the paper. All of the examples were based on existing and already proved Event-B developments. The table below demonstrates the number of proof obligations discharged automatically and manually to give a feeling of the cost of using flows in a development.

model	*total E.-B*	*auto E.-B*	*manual E.-B*	*total F.*	*auto F.*	*manual F.*
doors	15	14	1	12	12	0
doors1	26	26	0	10	7	3
doors2	8	8	0	18	15	3
doors3	14	14	0	10	10	0
filecopy	12	12	0	2	1	1
lift	36	30	6	7	7	0

In the table, " E.-B" stands for Event-B and "F." for flow; "auto" are the proof obligations discharged automatically while "manual" ones required some work in an interactive prover. Note that the table does not reflect the fact that some model changes were necessary when proving flow consistency. In most situations it was simply a case of a missing guard or invariant but, interestingly, some refinement steps had to be altered significantly to permit formulation of an interesting flow. This indicates that flows introduce some bias into model construction. We plan to investigate this subject further.

6 Conclusions

In our view, the ability to reason about event ordering is a useful addition to the Event-B method. It helps to construct models with rich control flow properties and it also makes such models more readable. Unlike the existing work in this area, it relies solely on theorem proving. It uses practical and scalable proof obligations that are handled well by automated theorem provers. The approach benefits from the existing tool support with a proof-of-the-concept tool implemented for the RODIN platform [14].

We attempted to solve the problem of unmanageable proofs resulting from a sequential composition of actions. For instance, in Classical B, actions within operations and events may be composed using operator ;, e.g., $a := a + 1; b := a + 1$. This is interpreted as applying the second action in the context of the first one. Unfortunately, the verification of sequential action composition is not compositional and all the composed actions must be analysed as a single logical statement. With flows, we make use of event guards to do localised reasoning where possible. In fact, in all the case studies attempted so far, it was possible to show flow consistency by strengthening event guards and adding new invariants with most of the proof obligations discharged automatically. This is despite the fact that in some example there were rather long chains of sequentially composed events (14 for the final refinement of the sluice control). The role of guards in analysing flow consistency is similar to the use of assertions in VDM [16] and refinement calculus [17]. Yet in our case, guards retain their primary role in the analysis of event feasibility, invariant preservation and refinement.

We have presented a three-step verification approach where one first establishes independently the well-formedness of a flow and consistency (and possibly refinement) of a machine and then checks the consistency of a machine and flow combination. In addition to the consistency condition, there is a possibility to generate proof obligations that would ensure that a flow is suitable for deriving an executable program. We are investigating some additional proof obligations. For example, we would like to be able, at least for some models, to establish the full equivalence of machine traces and flow traces. This would allow to reason about liveness and reachability properties without an assistance from a model checker.

The introduction of a flow is an intermediate step on the way to generating sequential program code from Event-B models. The addition of a flow to a machine converts an event-triggered, data-driven Event-B model into a sequential, control driven one. It is fairly straightforward to generate code from a combination of a concrete flow and a machine provided the issues of translating the Event-B mathematical language and abstract data types are resolved. It is possible that flows could play the role of B0 intermediate language [4] of Classical B for the Event-B method. However, instead of a single step transition into in implementable specification language flows permit a gradual detalisation using refinement. It remains to be investigated how successful flows mechanism would be in this role.

One direction for further research is finding a way of finer level of integration of flow and Event-B machine. As is stands, it is difficult to refactor an existing development by changing a some intermediate refinement step without changing its abstraction. In some cases, the control flow information is stored in auxiliary variables and if these variables are also mentioned in invariants it is impossible to remove them without also refactoring abstract models. At the same time, from our experience, in high-level models it is often easier to work with auxiliary variables rather than flow expressions. The possibility of constructing a refinement step from auxiliary variables model to a model with a flow would make the approach more attractive.

Using auxiliary variables is the prevalent technique in defining event ordering in an event-based specification method. It is usually ad hoc although [18] discusses an approach to a disciplined use of program counters. There is a substantial amount of work based on the Morgan's [12] failure-divergence semantics for event-based systems discussing the integration of state-based and process-based formalisms [19–21, 8, 22]. Their main difference from our approach is that consistency analysis is carried out with a help of process algebraic reasoning.

Our flow language lacks many constructs found in notations like CSP and CCS. In particular there are no communication primitives. It would be hard to justify a message passing mechanism for a single machine but it becomes an interesting possibility should a flow be able to relate several machines. The combination of CSP and Classical B has been investigated in [20] while the CSP style message passing was used to compose Event-B machines[13].

References

1. J. R. Abrial and L. Mussat, "Introducing Dynamic Constraints in B," in *Second International B Conference*. LNCS 1393, Springer-Verlag, April 1998, pp. 83–128.
2. J.-R. Abrial, "Event Driven Sequential Program Construction," 2000, available at http://www.matisse.qinetiq.com.

3. C. Metayer, J. Abrial, and L. Voisin, Eds., *Rodin Deliverable D7: Event B language*. Project IST-511599, School of Computing Science, Newcastle University, 2005.
4. J. R. Abrial, *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 2005.
5. H.Treharne and S.Schneider, "How to Drive a B Machine," 2000, pp. 188–208.
6. M.Butler and M.Leuschel, "Combining CSP and B for Specification and Property Verification," 2005, pp. 221–236.
7. C. Fischer and H. Wehrheim, "Model-Checking CSP-OZ Specifications with FDR," in *IFM '99: Proceedings of the 1st International Conference on Integrated Formal Methods*, K. Araki, A. Galloway, and K. Taguchi, Eds. London, UK: Springer-Verlag, 1999, pp. 315–334.
8. J. Woodcock and A. Cavalcanti, "The Semantics of Circus," in *ZB '02: Proceedings of the 2nd International Conference of B and Z Users on Formal Specification and Development in Z and B*. London, UK: Springer-Verlag, 2002, pp. 184–203.
9. R.-J. Back and K. Sere, "Stepwise Refinement of Action Systems," in *Proceedings of the International Conference on Mathematics of Program Construction, 375th Anniversary of the Groningen University*, J. L. A. van de Snepscheut, Ed. London, UK: Springer-Verlag, 1989, pp. 115–138.
10. C. A. R. Hoare, "Communicating Sequential Processes," *Commun. ACM*, vol. 21, no. 8, pp. 666–677, 1978.
11. M. Butler, "A CSP Approach to Action Systems. phd thesis." 1992.
12. C. Morgan, "Of wp and CSP," pp. 319–326, 1990.
13. M. Butler, "Decomposition Structures for Event-B," in *Integrated Formal Methods iFM2009, Springer, LNCS 5423*, vol. LNCS, no. 5423. Springer, February 2009.
14. "Event-B and RODIN Platform," http://www.event-b.org, 2004.
15. M. Leuschel and M. Butler, "ProB: A model checker for B," in *FME 2003: Formal Methods*, ser. LNCS 2805, K. Araki, S. Gnesi, and D. Mandrioli, Eds. Springer-Verlag, 2003, pp. 855–874.
16. C. B. Jones, *Systematic software development using VDM*. Prentice Hall International (UK) Ltd., 1986.
17. R.-J. J. Back and J. V. Wright, *Refinement Calculus: A Systematic Introduction*. Springer-Verlag New York, Inc., 1998.
18. M. J. Butler, "Event Ordering in Action Systems," in *Proc. Int. Refinement Workshop / Formal Methods Pacific'98, Springer Series in Discrete Mathematics and Theoretical Computer Science*, J. Grundy, M. Schwenke, and T. Vickers, Eds. Springer-Verlag, Berlin, 1998, pp. 61–80.
19. M. Leuschel and M. Butler, "Combining CSP and B for Specification and Property Verification," A. T. John Fitzgerald, Ian Hayes, Ed. Springer-Verlag, LNCS 3582, January 2005, pp. 221–236.
20. M. J. Butler, "An Approach to the Design of Distributed Systems with B AMN," in *Proc. 10th Int. Conf. of Z Users: The Z Formal Specification Notation (ZUM), LNCS 1212*, J. Bowen, M. Hinchey, and D. Till, Eds. Springer-Verlag, Berlin, April 1997, pp. 223–241.
21. S. Schneider, , S. Schneider, and H. Treharne, "Verifying Controlled Components," in *In Proc. IFM*. Springer, 2004, pp. 87–107.
22. C. Fischer, "CSP-OZ: a combination of object-Z and CSP," in *FMOODS '97: Proceedings of the IFIP TC6 WG6.1 international workshop on Formal methods for open object-based distributed systems*. London, UK, UK: Chapman & Hall, Ltd., 1997, pp. 423–438.