UDC 004.415.5

A. ILIASOV

School of Computing Science, Newcastle University, England

ON COMBINING EVENT-B AND WORKFLOW

In a state-based modelling method – Event-B, the next event to be executed is selected non-deterministically among all the currently enabled events. The information about event ordering has to be embedded into guards and before-after predicates of events. This results in entanglement of control flow and functional specification plus the addition of extra variables resulting increased complexity and reduced readability of a model. In this paper we consider Event-B and discuss how to complement the Event-B development process with a workflow view. Proposed approach combines the elements of a workflow language and state-based modelling.

Key words: *formal modelling, Event-B, workflow.*

Introduction

Application of formal modelling is important to engineer safe and dependable systems. Using a formal method, however, is not always easy. One of the reasons is the fact that a modelling method dictates a specific viewpoint on a system, as required by the conceptual framework of a method. At times, such a viewpoint does not fit well a modeller's view and intuition about a system, thus hindering or preventing the system modelling.

We consider a state-based modelling method – Event-B [1, 2] and discuss how to complement the Event-B development process with a workflow view. The approach assists in constructing models and carrying out refinement by employing the visional intuition of a workflow language. It although facilitates the reasoning about models for which a high-level workflow description is natural.

The approach combines the elements of a workflow language and state-based modelling. The workflow part describes the control flow of a system while the state part details state evolution. System state is described by a collection of variables constrained by an invariant. State is transformed with the help of events. In addition to the ordering imposed by the workflow part of a model, an events also carriers a guard expressing enabledness constrains as a predicate on system variables. A workflow in this proposal is a simple process algebra constraining the order of events execution in an Event-B model.

2. Event-B Overview

An Event-B model characterises the state of a system at any given moment and the way the system would evolve further from that state. Mathematically, an Event-B model is a state-mapping predicate defined by a set events that are relations on the new and old system states. An event is characterised by a guard, describing the set of states from which an event may be executed, and a post-condition, defining the states upon the event execution. In addition, system states are constrained with an invariant expressing the properties to be maintained throughout the system lifetime. A system state is given by a collection of named and typed variables.

An Event-B model, called a machine, has the following general form:

SYSTEM name
VARIABLES V
INVARIANT I
INITIALISATION Ri
EVENTS
e_1 = …
…
e_k = …

where name is a model name, v is a vector of model variables, I(v) is a model invariant defined on model variables v, Ri(v') is an initialisation event computing an initial model state (with referring to a previous, non-existent, state v).

Model events e_1 ..., e_n have the following structure: e_i = **when** G(v) **then** S(v, v') **end**. Here e_i is an event name; G(v) is a guard predicate defining the states when the event is enabled; S(v,v') is a before-after predicate relating a previous state v with a new state v'.

This is a simplified version of Event-B syntax. The complete syntax has can be found in [1].

Theorem proving is the primary formal verification technique for Event-B models. Proof obligations are generated automatically and passed on to a collection of

mechanised theorem provers. In most cases, automated theorem proving would discharge the majority of proof obligations but the remaining are to be proved with an assistance of a modeller in an interactive proving environment [3]. A model checking tool is available as well [4].

3. Workflow

In Event-B, the next event to be executed is selected non-deterministically among all the currently enabled events. The information about event ordering has to be embedded into guards and before-after predicates of events. This results in entanglement of control flow and functional specification plus the addition of extra variables resulting increased complexity and reduced readability of a model. One way to decouple control flow from the rest of a specification is by defining the control flow information in a dedicated expression imposing ordering on actions.

3.1. A Workflow Language

An entity used to describe control flow information is called a workflow. A workflow is contained in a machine and is formulated separately from the descriptions of events.

One way to reason about a workflow is to treat it as a partial order on events. Hence, a workflow can be characterised by a function mapping some current event into the set next of events:

$$F : A \rightarrow\!\!\!\!\!+ \; P(A)$$

where A is the set of events of a machine. Such a function is conveniently constructed using the following simple process algebraic notation:

$skip_i$	implicit stuttering step with index i
e_i	action a with index i
p; q	sequential composition
p Π q	non-deterministic choice
loop(p)	loop

skip corresponds to an implicit event which does not modify a model state. To distinguish between different instances, each occurrence of **skip** in a workflow expression is attached a unique index. Similarly, a workflow action is composed from a name of an event and index, distinguishing the occurrences of the same event in a workflow expression.

The traces of a workflow, considered in isolation, are given by function tr, defined below:

$$tr(skip) = \left\{\langle\rangle, \langle\surd\rangle\right\};$$

$$tr(a_i) = \left\{\langle\rangle, \langle a_i\rangle, \langle a_i, \surd\right\};$$

$$tr(pq) = \left\{s \cap t \mid s \cap \langle\surd\rangle \in tr(p) \wedge t \in tr(q)\right\} \cup$$

$$\cup \left\{s \cap \langle e\rangle \mid s \cap tr(p) \wedge e \neq \surd\right\};$$

$$tr(p\Pi q) = tr(p) \cup tr(q);$$

$$tr(loop(p)) = tr((p; loop(p))\Pi skip).$$

Here <a,..., b> is a trace characterised by an ordered set and V marks a successful termination of a workflow. A trace terminating with anything but V describes a trace leading to deadlock. The ^ operator extends a trace with another trace: (s1 ..., sn) ^ {t1,..., tm) = {s1,...,sn, t1,...,tm).

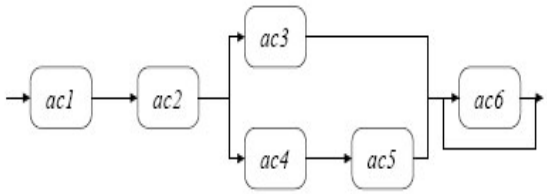

Fig. 1: A graphical representation of workflow expression aci;ac2; (ac3Π(ac4; ac5)); loop(ac6)

The F function, defined above, is computed for some workflow expression wf by analys the expression traces and building the list of all actions which may follow a given action a_i:

$$F(a_i) =$$

$$= \left\{n \mid hd \cap \langle a_i\rangle \cap \langle n\rangle \cap tl \in tr(wf) \wedge n \neq \left\{\surd\right\}\right\}.$$

The function is undefined for the termination mark V.

The coupling of the workflow with module state is achieved by changing the tr (ai) rule to make it deliver an empty trace for the states where the guard of action a is not enabled.

The reference to a current state δis recorded along with a trace and is passed as an argument to tr:

$$\begin{array}{lcl}
tr(\delta, \mathbf{skip}) & = & \{(\delta, \langle\rangle), (\delta, \langle\surd\rangle)\}) \\
tr(\delta, a_i) & = & \{(\delta, \langle\rangle)\}, \text{and } \neg G_a(\delta) \\
tr(\delta, a_i) & = & \{(\delta, \langle\rangle), (a(\delta), \langle a_i\rangle), (a(\delta), \langle a_i, \surd\rangle)\}, \text{and } G_a(\delta) \\
tr(\delta, p; q) & = & \{s \frown t \mid s \frown (\sigma, \langle\surd\rangle) \in tr(\delta, p) \wedge t \in tr(\sigma, q)\} \cup \\
 & & \{s \frown (\sigma, \langle e\rangle) \mid s \frown (\sigma, \langle e\rangle) \in tr(\delta, p) \wedge e \neq \surd\} \\
tr(\delta, p \sqcap q) & = & tr(\delta, p) \cup tr(\delta, q) \\
tr(\delta, loop(p)) & = & tr(\delta, p; loop(p)), \text{and } \bigvee_{a \in EN(p)} G_a(\delta) \\
tr(\delta, loop(p)) & = & tr(\delta, \mathbf{skip}), \text{and } \neg \bigvee_{a \in EN(p)} G_a(\delta)
\end{array}$$

Here Ga stands for the guard of an event a associated with an indexed action ai. An event name a plays a dual role. It is recorded along with its index in a trace to mark an occurrence of the event in the execution history. It also acts as a notational shortcut for the state transformer implemented by the event and used to compute a new state from some current state (the second and third rules).

Reasoning about the properties of a workflow requires the notions of entry and exit workflow points. An entry point of a workflow is an action initially enabled when a workflow expression starts execution.

Similarly, an exit point is one of the possible termination actions. The sets of entry and exit points of a workflow are computed by functions EN (entry points) and EX (exit points), defined as follows:

$$\begin{array}{lcllcl} EN(e) & = & \{e\} & EX(e) & = & \{e\} \\ EN(p;q) & = & EN(p) & EX(p;q) & = & EX(q) \\ EN(p \sqcap q) & = & EN(p) \cup EN(q) & EX(p \sqcap q) & = & EX(p) \cup EX(q) \\ EN(loop(p)) & = & EN(p) & EX(loop(p)) & = & EX(p) \end{array}$$

An Event-B model without a workflow is equivalent to one having the following workflow attached:

$$loop(\sqcap_{e \in E} e),$$

where E is the set of events of a machine. The workflow expression states that events are executed as long as there is at least one enabled event left.

3.2. Interval and Point Constraints

A global invariant, characterising the global module properties, must be maintained by every state-transforming rule of a module. It may e also needed to reason about properties of model fragments. Such a property would clarify and document model parts or even drive the development process by borrowing from system requirements. Such additional knowledge about model fragments provides an assistance in model correctness proofs.

One way to reason about local properties is to require a model to satisfy a given property at a certain point of its history. Such property is called a point constraint. Unlike a global invariant, characterising a model state irrespective to the execution history, a point constraint characterises a slice of states associated with the specific histories of model execution. Evidently, for some fixed point in a system history, it is possible to formulate conditions that are stronger than those spanning the whole system history.

States associated with a point constraint are identified by extending the workflow expression notation. A point constraint is marked in a workflow by a predicate placed within square brackets: [P(v)].

An action preceding a point constraint must establish the condition expressed in the constraint. This is results in a new obligation similar to that of invariant preservation [1]. Symmetrically, an event following a point constraint may rely on the condition expressed in the invariant2. The notions of predecessor and successor are defined on the traces of a workflow. An action precedes a point invariant if a silent action corresponding to the point invariant follows the action in at least one of the workflow traces. An action follows a point invariant if the point invariant precedes the action in every workflow trace.

A logical extension of the point constraint concept is interval constraint. An interval constraint, as suggested by its name, must be maintained during an interval of a workflow trace. An interval constraint is declared by attaching a predicate, expressing a desired property, to a workflow sub-expression: [P(v)](wf). On the preceding and following actions, an interval constraint has the same effect as a point constraint. Additionally, an interval constraint must be satisfied on all the traces of the associated workflow expression.

3.3. Workflow Verification

Workflow/Guard Consistency It is required to demonstrate that the combination of a workflow and events is not contradictory and for every action in a workflow there exists an enabled event among the set of the next events computed by the workflow.

$$\forall_{a_i}(a_i \in dom(F) \wedge I(v) \wedge G_a(v) \wedge S_a(v,v') \Rightarrow \exists_{n_j \in F(a_i)} G_n(v')).$$

The above is equivalent to the following:

$$\bigwedge_{a_i \in dom(F)} (I(v) \wedge G_a(v) \wedge S_a(v,v') \Rightarrow \bigvee_{n_j \in F(a_i)} G_n(v'))$$

K&dom{F)(I(v) Ga(V) A Se(t,,t/) =* Vn^FCai) ^')).

The statement above, formulated for some indexed action a*, can be summarised as follows WF.GRD

$$I(v) \wedge G_a(v) \wedge S_a(v,v') \Rightarrow \bigvee_{n_j \in F(a_i)} G_n(v'),$$

which says that the after-state of an event a must imply the disjunction of guards of the next events offered by the workflow for this particular occurrence of the action in the workflow expression.

Such rely, however, is conditional itself since an event may occur in several places in a workflow and thus not always it may benefit from a point constraint. On the other hand, a point constraint must be satisfied by an event even when not all of its occurence precede a point constraint clause.

Interval and Point Constraints The semantics of point and interval constraints is defined by the way they affect the before and after state of an event. The following condition must be satisfied for an event e which is associated with an action followed by a point constraint Pe:

$$Q(s,c) \wedge I(c,s,v) \wedge G_e(c,s,v) \wedge S_e(c,s,v,v') \Rightarrow P_e(c,s,v').$$

For an interval constraint, such obligation is generated for each action occurring in the attached workflow expression. An event associated with an action immediately following a point or interval constraint Re or an action associated with an interval constraint Re assumes the constraint condition whenever it is enabled. Since an event may be associated with more than one

action with different point or interval constraints and possibly no constraints at all the condition that an event may rely on is the disjunction of all such constraint conditions. An absence of any constraint is treated as a constraint with a copy of the model invariant.

$$Q(s,c) \wedge I(c,s,v) \wedge G_e(c,s,v) \Rightarrow R_e^1(c,s,v) \vee \cdots \vee R_e^k(c,s,v).$$

Combining the above, the following general form is obtained WF_INV

$$Q(s,c) \wedge I(c,s,v) \wedge G_e(c,s,v) \wedge (R_e^1(c,s,v) \vee \cdots \vee R_e^k(c,s,v)) \wedge S_e(c,s,v,v') \Rightarrow P_e(c,s,v'),$$

where Pe is the property to be established to satisfy a following point or interval constraint.

Rule WF_INV may be used to propagate a property through a workflow expression without introducing additional variables and invariants. Property propagation may be needed to establish the cumulative effect a group of events.

Convergence With the addition of the workflow concept, the Event-B convergence condition ([1]) can be reformulated. A workflow whose traces terminate with V everywhere automatically imposes the convergence condition. In fact, only the non-convergence of loop may result in the non-convergence of the containing action group3.

Thus, to demonstrate the convergence of an action group it is enough to independently demonstrate the convergence of each loop construct that brings the benefit of being able to formulate a variant independently for each new loop construct. The conv defines the convergence computation rules:

$$\begin{aligned}
conv(\text{skip}) &= true \\
conv(p;q) &= conv(p) \wedge conv(q) \\
conv(p \sqcap q) &= conv(p) \wedge conv(q) \\
conv(loop(p)) &= conv(p) \wedge \exists \text{var} \cdot (\text{var}(v) \in \mathbb{N} \wedge \forall a \in EN(p) \cdot (\\
&\qquad Q(\ldots) \wedge I(\ldots) \wedge G_a(\ldots) \wedge S_a(\ldots) \Rightarrow \text{var}(v') < \text{var}(v)))
\end{aligned}$$

The convergence of a loop requires the convergence of its body and the existence of a common well-founded variant for the entry points of the loop body.

The termination condition - the fact that a workflow always progress towards a successful termination - is a corollary of the Workflow/Guard Consistency and Convergence. Indeed, the former guarantees an unfailing workflow progress while the later establishes that such progress has its end.

This matches the informal interpretation of new events with non-trivial variants as loops in an Event-B model.

Discussion

The most strongly related works are those of combining CSP [5] and B. In [6] B models are controlled by CSP processes. A B model is seen as a passive collection of event and the CSP part completely control the order of event execution. Since a B model is passive, it may not contain guarded command - only operations with preconditions are allowed. It is not clear how this approach could applied to Event-B which is based exclusively on guarded events. Another closely related work is that of [7] that discusses a way to transform a combination of a subset of CSP and B into a plain B model.

The downside of the approach is that a resulatnt B model would contain generation artefacts that could make theorem proving tricky. [8] is a model checker for a combination of B and CSP models. It this approach a B model is treated as a CSP process and a model checker verifies a parallel composition of a B machine and a CSP expression.

Much work has been done in integrating the popular Z formalism with CSP. One of the more prominent examples is the hybrid Circus modelling language [9] which borrows from both Z and CSP and offers a refinement based development process. There are also a number of works proposing verification mechanism for a combination of a Z and a CSP models. CSP-Z [10] translates a Z model into CSP to benefit from the availability of high-quality CSP model checkers; [11] discusses a model checking approach for a combination of an Object Z and CSP.

References

1. Metayer C Rodin Deliverable D7: Event B language. / C. Metayer, J. Abrial, L. Voisin. - Project IST-511599. - School of Computing Science, Newcastle University. - 2005.

2. Abrial J.R.: The B-Book: Assigning Programs to Meanings. / J.R. Abrial. - Cambridge University Press. - 2005.

3. RODIN Event-B Platform. [Электронный ресурс]. – Режим доступа к ресурсу: http://rodin-b-sharp.sourceforge.net/.

4. Leuschel M. Prob: A model checker for b. In: FME 2003 / M. Leuschel, M. Butler // Formal Methods, LNCS 2805. - Springer-Verlag. - 2003. - P. 855-874.

5. Hoare C.A. Communicating Sequential Processes. / C.A. Hoare // Commun. ACM 21(8). -1978. - P. 666-677.

6. Treharne H. How to Drive a B Machine. / H. Treharne, S. Schneider. - 2000. - P. 188-208.

7. Butler M.J. A practical approach to combining csp and b. / M.J. Butler // In: FM '99: Proceedings of the Wold Congress on Formal Methods in the Development of Computing Systems. –Vol. I. -London, UK, Springer-Verlag. - P. 490-508.

8. Butler M. Combining CSP and B for Specification and Property Verification / M.Butler, M.Leuschel: - 2005. - P. 221-236.

9. Woodcock J. The semantics of circus / J. Woodcock, A. Cavalcanti // 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. - P. 184-203.

10. Mota A. Model-checking csp-z: strategy, tool support and industrial application / A. Mota, A. Sampaio // Sci. Comput. Program. - 40(1). - 2001. - P. 59-96.

11. Fischer C. Csp-oz: a combination of object-z and csp / C. Fischer // 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. - P. 423-438.

Поступила в редакцию 15.02.2009

Рецензент: д-р техн. наук, проф. Б.М. Конорев, Национальный аэрокосмический университет им. Н.Е. Жуковского «ХАИ», Харьков, Украина.

ОБЪЕДИНЕНИЕ EVENT-B И ПОТОКА ПОСЛЕДОВАТЕЛЬНОСТИ ВЫПОЛНЯЕМЫХ ДЕЙСТВИЙ

А. Ильясов

В методе моделирования Event-B, следующее событие, которое должно выполниться, выбирается случайным образом из всех доступных событий. Информация о порядке выполнения событий должна быть заложена в предикатах событий с предусловиями и постусловиями. Результатом этого является запутанность потока управления и функциональной спецификации, к тому же увеличивается число дополнительных переменных и растет сложность, а также понижается читабельность модели. В статье рассматривается Event-B и обсуждается вопрос расширения процесса разработки Event-B с точки зрения потока последовательности выполняемых действий. Предложенный подход объединяет элементы языка потока и моделирования, основанного на состояниях.

Ключевые слова: формальное моделирование, Event-B, поток последовательности выполняемых действий.

ОБ'ЄДНАННЯ EVENT-B ТА ПОТОКУ ПОСЛІДОВНОСТІ ДІЙ ЩО ВИКОНУЮТЬСЯ

А. Ільясов

В методі моделювання Event-B, наступна подія, яка повинна виконуватись, вибирається випадковим чином із усіх доступних подій. Інформація про порядок виконання подій повинна бути закладена у предикатах подій з передумовами та посту мовами. Результатом цього є заплутаність потоку керування і функціональної специфікації, крім того збільшується кількість додаткових змінних і зростає складність, а також зменшується читабельність моделі. У статті розглянуті питання розширення процесу розробки Event-B з погляду потоку послідовності дій що виконуються. Запропонований підхід об'єднує елементи мови потоку і моделювання, що базується на станах.

Ключові слова: формальне моделювання, Event-B, потік послідовності дій що виконуються.

Алексей Ильясов – канд. техн. наук, ассоциированный исследователь, School of Computing Science, Newcastle University, England, e-mail: Alexei.Iliasov@ncl.ac.uk.