Structured Event-B Models and Proofs

Corrected Version

Stefan Hallerstede

University of Düsseldorf, Germany, `stefan.hallerstede@wanadoo.fr`

Abstract. Event-B does not provide specific support for the modelling of problems that require some structuring, such as, local variables or sequential ordering of events. All variables need to be declared globally and sequential ordering of events can only be achieved by abstract program counters. This has two unfortunate consequences: such models become less comprehensible — we have to infer sequential ordering from the use of program counters; proof obligation generation does not consider ordering — generating too many proof obligations (although these are usually trivially discharged).
In this article we propose a method for specifying structured models avoiding, in particular, the use of abstract program counters. It uses a notation that mainly serves to drive proof obligation generation. However, the notation also describes the structure of a model explicitly. A corresponding graphical notation is introduced that visualises the structure of a model.

1 Introduction

Recently, we have argued that the benefits of the minimalist approach of Event-B [1] to formal modelling are sometimes balanced by complications that result, in particular, from more complicated invariants [11]. The reason for this is the necessity to introduce abstract program counters when dealing with models that require (sequential) ordering of some events. However, we have argued in an earlier article [9] that, specifically, structuring constructs like sequential composition or if-statements lead to complications. Thus, the problem we face is keeping the simplicity resulting from the minimalism while providing some means to structure Event-B models. The solution we propose is to move more information about what is to be proved into the models — a solution we have already chosen before by introducing witnesses to Event-B. We do not introduce sequential composition or if-statements but a notation that allows us to state properties to prove about them. The usual approach in program verification would be to derive proof obligations from a program following its structure. We do not have a program but work exclusively with the proof obligations.

We need to specify control flow in Event-B models without having to resort to implementing abstract program counters. In principle proof outlines [16] can accomplish this. However, similarly to [14], we want to avoid introducing a concrete syntax of a programming notation.

Proof outlines are a compact representation of correctness proofs using Hoare triples [12]. A Hoare Logic states what is to be proved about a program S. For predicates p and q we write $\{p\}\,S\,\{q\}$ to state that "starting from a state satisfying p program S leads to a state satisfying q". Sequential composition of programs S and T is proved to satisfy $\{p\}\,S\,;T\,\{q\}$ by a rule

$$\frac{\{p\}\,S\,\{r\} \quad \{r\}\,T\,\{q\}}{\{p\}\,S\,;T\,\{q\}}$$

Proof outlines [16] represent this more succinctly,

$$\{p\} \quad S\,; \quad \{r\} \quad T \quad \{q\}$$

annotating the program $S\,;T$ with all predicates involved in the proof. This notation is used extensively in [4] to present correctness proofs of programs.

Similarly to proof outlines, temporal verification diagrams [14] specify alternating sequences of assignments and assertions. In addition, they provide hierarchical structuring based on state charts. Certain "patterns" of diagrams are identified that are instrumental in proofs of temporal properties of reactive systems. We combine ideas of [16] and [14] in our proposal for structured Event-B.

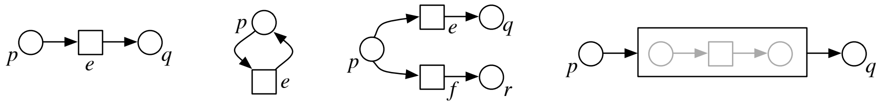
Overview. We introduce structured Event-B in Section 2. A small example in Section 3 describes the relationship between Event-B and structured Event-B. In Section 4 we develop a simple sequential program to show how the method could be used in practice. Section 5 points to related and future work and Section 6 contains a conclusion.

2 Event-B with Structure

We introduce a structuring notation for Event-B that maintains the simplicity of the original Event-B proof obligations. In this article, we identify two concepts that are missing from Event-B: sequentiality and locality. Event-B is strongest at proving properties of highly concurrent systems that mostly use global variables. Our structured notation supports sequentiality and locality while keeping the ease of use of Event-B. Concurrency can be modelled explicitly in the style of [16]. Here, we focus on sequentiality as this is at the moment difficult to model in Event-B.

2.1 Notation

Before introducing the notation in detail, we provide some small examples of terms of the notation together with a graphical notation that we use for illustration. (The graphical notation is not an exact representation. It's main purpose is to clarify a model. See also [6].) The structure notation is based on assertions p, q and r, and events e and f. We write $p \rightarrow e \rightarrow q$ for "starting from assertion p event e establishes q", Figure 1a; we write $p \rightarrow e -$ for "assertion p is an invariant of event e”, Figure 1b, and $p \to e \leftarrow$ if e is convergent using the same graphical representation; we write $p \to (e \to q \mathbin{[\!]} f \to r)$ for “starting from p event e establishes q or event f establishes r”, Figure 1c; and we write $p \to [\,S\,] \to q$ for “starting from p box S establishes q” where S is any term, Figure 1d. In Section 5



(a) Step (b) Loop (c) Choice (d) Box

Fig. 1: Graphical representation of the structure notation

we describe briefly a construct for concurrency that we are considering. For the purpose of this article the notation outlined above is sufficient.

In order to define some operators on structure terms, we need to know about their possible shapes. The syntax of the (sequential) structure notation (p, q are predicates, and e is an event that may be decorated with a “!”, see below) is:

$$\begin{array}{lcl}
S & ::= & p \to T \\
T & ::= & U \to S \;\mid\; U \to q \;\mid\; U \,- \;\mid\; e \leftarrow \;\mid\; T_1 \mathbin{[\!]} T_2 \\
U & ::= & e \;\mid\; [\,S\,]
\end{array}$$

We define two operators $\mathcal{I}$ and $\mathcal{F}$ on the syntax of the structure notation yielding the *initial assertion* and the *final assertions* of a term, respectively. The initial assertion is defined by $\mathcal{I}(p \to T) \;\widehat{=}\; p$. The final assertion of a term is the disjunction of the “end points” of the term,

$$\begin{array}{lcl@{\qquad}lcl}
\mathcal{F}(p \to U \to S) & \widehat{=} & \mathcal{F}(S) & \mathcal{F}(p \to U\, -) & \widehat{=} & \mathit{false} \\
\mathcal{F}(p \to U \to q) & \widehat{=} & q & \mathcal{F}(p \to e \leftarrow) & \widehat{=} & \mathit{false} \\
\mathcal{F}(p \to (T_1 \mathbin{[\!]} T_2)) & \widehat{=} & \mathcal{F}(p \to T_1) \;\vee\; \mathcal{F}(p \to T_2) & & &
\end{array}$$

The definition of the final assertion is consistent with the axiomatic semantics of non-deterministic choice [4]. This will become apparent in the description of assertions below. The intuition behind the definition of $\mathcal{F}(p \to U\, -)$ and $\mathcal{F}(p \to e \leftarrow)$ is that the event “returns to p”; and in absence of an exiting choice, for example, $p \to (U_1 - \mathbin{[\!]} U_2 \to q)$, it does not have an “end point”.

2.2 Proof Obligations

Proof obligations are defined following the structure notation. In fact, the main purpose of the structure notation is to drive the generation of proof obligations in a more evident manner.

Assertions. To state the proof obligations for assertions we consider all suitable sub-terms of a structure term. For instance, given the following structure term

$p(v) \rightarrow e(v) \rightarrow q(v) \rightarrow f(v) \rightarrow r(v)$, we consider the sub-terms $p(v) \rightarrow e(v) \rightarrow q(v)$ and $q(v) \rightarrow f(v) \rightarrow r(v)$.

Let $e(v)$ be an event with guard $g(v)$ and action $v :| \ a(v, v')$. For the sub-term $p(v) \rightarrow e(v) \rightarrow q(v)$ we prove

$$p(v) \land g(v) \land a(v, v') \Rightarrow q(v') \qquad \textit{assertion preservation}$$

We also prove *action feasibility*, $p(v) \land g(v) \Rightarrow \exists v' \cdot a(v, v')$, if an event is not refined further. Loops and choices are treated similarly to steps. Terms $p(v) \rightarrow e(v) -$ and $p(v) \rightarrow e(v) \leftarrow$ correspond to $p(v) \rightarrow e(v) \rightarrow p(v)$ but make explicit that $p(v)$ is invariant. The term $p(v) \rightarrow (e_1(v) \rightarrow q_1(v) \ [] \ e_2(v) \rightarrow q_2(v))$ corresponds to two terms $p(v) \rightarrow e_1(v) \rightarrow q_1(v)$ and $p(v) \rightarrow e_2(v) \rightarrow q_2(v)$.

For a box $p(v) \rightarrow [\, S(v) \,] \rightarrow q(v)$ we have to prove

$$p(v) \Rightarrow \mathcal{I}(S(v)) \qquad \textit{box entry}$$
$$\mathcal{F}(S(v)) \Rightarrow q(v) \qquad \textit{box exit}$$

Convergence. The term $p(v) \rightarrow e(v) \leftarrow$ states that p is an invariant of $e(v)$ and that $e(v)$ is *convergent*, that is, it decreases a variant. If a variant $t(v)$ is specified for $e(v)$ or a refinement of $e(v)$, denoted by $e(v) \leftarrow t(v)$, we prove convergence of $e(v)$ in terms of *variant boundedness*, $p(v) \land g(v) \Rightarrow t(v) \geq 0$, and *variant progress*, $p(v) \land g(v) \land a(v, v') \Rightarrow t(v') < t(v)$. In unstructured Event-B events can be declared to be anticipated in order to delay a convergence proof to some refinement. In structured Event-B convergence is proved only when a variant is stated which may happen in a refinement. So the distinction between convergent and anticipated events disappears.

Refinement. We consider three forms of refinement, *structure* refinement, *event* refinement, and *box* refinement. A structure refinement replaces an event in a refined model by a structure. Event refinement relates two events, box refinement two boxes. Structure refinement is defined in terms of event and box refinement. An event $e(v)$, occurring in a term $p(v) \rightarrow e(v) \rightarrow q(v)$, is structure refined by a term $R(v, w)$, denoted by $e(v) \ \sim \ R(v, w)$, where the term $R(v, w)$ must contain at least one event decorated with an exclamation mark. The assertions $p(v)$ and $q(v)$ are associated with gluing assertions $p^*(v, w)$ and $q^*(v, w)$. We prove

$$p(v) \land p^*(v, w) \Rightarrow \mathcal{I}(R(v, w)) \qquad \textit{box entry}$$
$$\mathcal{F}(R(v, w)) \Rightarrow q^*(v, w) \qquad \textit{box exit}$$

Note that $p^*(v, w)$ needs to be established by the event that refines the event preceding $e(v)$ in the abstract term. We do not allow strengthening of assertions in any other case.

Two kinds of events occur in $R(v, w)$, decorated events $f(w)!$ that refine event $e(v)$ and, undecorated, new events $f(w)$ that refine *skip*. Let $h(w)$ be the guard of $f(w)$ and $w :| \ b(w, w')$ its action. The predicate $m(v, w, v', w')$ denotes *witnesses* for the abstract variables v' linking abstract variables to concrete variables. Witnesses describe for each event separately how the refinement is achieved

[9]. For decorated events $f(w)!$ occurring in a term $r(v,w) \rightarrow f(w)! \rightarrow s(v,w)$, let $\phi(v,w,w') = r(v,w) \wedge h(w) \wedge b(w,w')$; we prove

$\phi(v,w,w') \Rightarrow \exists v' \cdot m(v,w,v',w')$	*witness feasibility*
$\phi(v,w,w') \Rightarrow g(v)$	*guard strengthening*
$\phi(v,w,w') \wedge m(v,w,v',w') \Rightarrow a(v,v')$	*action simulation*
$\phi(v,w,w') \wedge m(v,w,v',w') \Rightarrow s(v',w')$	*assertion preservation*

For undecorated events $f(w)$ occurring in a term $r(v,w) \rightarrow f(w) \rightarrow s(v,w)$ we prove that they refine *skip*; we prove *assertion preservation* $\phi(v,w,w') \Rightarrow s(v,w')$.

Box refinement maintains the box-entry property once proved. A box $[\,S(v)\,]$, occurring in a term $p(v) \rightarrow [\,S(v)\,] \rightarrow q(v)$, is refined by a box $[\,R(v,w)\,]$ where $S(v)$ and $R(v,w)$ are identical terms except for assertions contained in $R(v,w)$ that may be strengthened. Box refinement is established by *box entry* and *box exit* proof obligations with respect to the gluing assertions $p^*(v,w)$ and $q^*(v,w)$.
Enabledness. Enabledness proof obligations in Event-B can be used to verify deadlock-freeness or precondition weakening [10], for instance. In structured Event-B the large disjunctions that would appear in Event-B enabledness proof obligations can be smaller depending on the structure term $R(v,w)$ of a structure refinement $e(v) \sim R(v,w)$. For terms $r(v,w) \rightarrow f(w) \rightarrow s(v,w)$, $r(v,w) \rightarrow f(w) -$, and $r(v,w) \rightarrow f(w) \leftarrow$ contained in $R(v,w)$ we prove $r(v,w) \Rightarrow h(w)$. For a choice term $r(v,w) \rightarrow (f_1(w) \rightarrow s_1(v,w) \;[\!]\; f_2(w) \rightarrow s_2(v,w))$ we prove $r(v,w) \Rightarrow h_1(w) \vee h_2(w)$. If $r(v,w)$ is the initial assertion of $R(v,w)$, the abstract guard $g(v)$ is added to the premise.

3 Event-B with and without Structure

The Event-B method as described in [3] has a certain structure that is not made formally explicit. However, it is mentioned in the informal description in [3]. Not taking into account convergence, this would correspond to

$$\mathit{true} \rightarrow \boldsymbol{initialisation} \rightarrow \mathit{inv} \rightarrow (\boldsymbol{event}_1 - [\!] \ldots [\!]\ \boldsymbol{event}_n -)$$

where *inv* is the invariant. The correspondence described in this section is not intended to suggest such a definition. It is serves merely to explain the structured notation in terms of the unstructured notation.

3.1 Without Structure

We give a very simple example of a structured model and a corresponding unstructured model. Being very simple, too, proofs are omitted. We model an abstract program that sets y to 2. In order to represent structure in unstructured Event-B we have to introduce an abstract program counter *apc*, say, with values *aini*, *aend*, yielding a model with invariant

$$\mathit{apc} = \mathit{aini} \Rightarrow y = 0$$
$$\mathit{apc} = \mathit{aend} \Rightarrow y = 2$$

and events

```
initialisation      convergent inc2
  apc := aini         when apc = aini then
  y := 0                apc := aend
                        y := y + 2
```

We would show convergence of ***inc2*** to show that the abstract program counter is modelled correctly (using the variant $\{apc\} \cap \{aini\}$, for instance).

We refine the abstract model by one that increments a variable x in two steps. We use two events ***incx1*** and ***incx2***,

```
initialisation      convergent incx1          convergent incx2
  cpc := aini         when cpc = aini then      when cpc = amid then
  x := 0                cpc := amid               cpc := aend
                        x := x + 1                x := x + 1
```

and a gluing invariant

$$cpc \in \{aini, aend\} \Rightarrow x = y$$
$$cpc \in \{aini, amid\} \Rightarrow apc = aini$$
$$cpc = amid \Rightarrow x = y + 1$$

that relates concrete variables x to abstract variables y depending on the value of the program counter. It is also necessary to relate the program counters apc and cpc by $cpc \in \{aini, amid\} \Rightarrow apc = aini$. Control flow is modelled explicitly.

3.2 With Structure

Using the structure notation, there is no need to model program counters. Assertions $aini$ and $aend$

$$@aini \quad y = 0$$
$$@aend \quad y = 2$$

(read: at $aini$ "$y = 0$") are stated at those locations where they hold

$$true \rightarrow \textbf{\textit{iniy}} \rightarrow aini \rightarrow \textbf{\textit{inc2}} \rightarrow aend$$

The control flow is modelled by the structure term. It is not represented in the formal text otherwise. The model contains events ***iniy*** and ***inc2***

```
iniy          inc2
  y := 0        y := y + 2
```

Similarly to the unstructured model, we refine the abstract model by incrementing twice. Event ***inc2*** is structure refined by the ***incx1*** and ***incx2***,[1]

$$\textbf{\textit{inc2}} \sim aini \rightarrow \textbf{\textit{incx1}} \rightarrow amid \rightarrow \textbf{\textit{incx2}}! \rightarrow aend$$

[1] We could also have used the same event twice. But in this article we want to keep the convention that each event appears only once. The structure refinement notation $e \sim R$ used in this article does not consider the position of e in the abstract term.

and the abstract event ***iniy*** by the concrete event ***inix*** which is stated formally ***iniy*** $\sim$ *true* $\rightarrow$ ***inix***! $\rightarrow$ *aini*.

inix	***incx1***	***incx2***
$x := 0$	$x := x + 1$	$x := x + 1$

With the gluing assertions

$@aini \quad x = y$
$@amid \quad x = y + 1$
$@aend \quad x = y$

we have to prove, for instance, that ***incx1*** event refines *skip* and ***incx2*** event refines abstract event ***inc2***

$$x = 0 \wedge x = y \Rightarrow x + 1 = y + 1$$
$$x = y + 1 \Rightarrow x + 1 = y + 2$$

These proof obligations correspond closely to the proof rule of the refinement calculus [15] for sequential composition.

3.3 Remarks

With the structure made explicit we can immediately see the sequencing in the model whereas in the unstructured model we have to look closely to see it. This becomes more convincing in larger examples like the one of Section 4, for example. In addition, the assertions of the structured model are simpler than the invariants of the unstructured model. In particular, it is not necessary to specify sequencing information relating only abstract program counters. The refinement proofs are not more difficult in the structured model although the formal definition is more complex mostly due to the box proof obligations. Note, however, that usually we do not have to prove anything at all for boxes. This is because we reuse assertions already declared, trivially satisfying the implications of box entry and box exit, for instance, *aini* and *aend* in the model above. We have removed one source of complexity: the choices to code the assertions in the invariant are no longer available. We believe this makes the method easier to use. The main drawback of the structured approach is that the refinement notion is more restrictive, being defined per event and no longer per model.

4 Development of a Sequential Program

We demonstrate the use of structured Event-B by means of a sequential program development, the extended Euclidian algorithm (Figure 2). The refinement steps are illustrated using the graphical notation of Figure 1. We believe it to be very useful for understanding the model more easily. For instance, the collapsed representation in Figure 3 of the final model of the sequential program shows nicely the control flow of the program. The example is large enough to illustrate

```
        when $b \neq 0$ then
upini:    $f, s, t, q, r := 0, \{0 \mapsto a\}, \{0 \mapsto b\}, \{0 \mapsto a \div b\}, \{0 \mapsto a \bmod b\}$;
          while $r(f) \neq 0$ do
up:         $f, s(f{+}1), t(f{+}1), q(f{+}1), r(f{+}1) := f{+}1, t(f), r(f), t(f) \div r(f), t(f) \bmod r(h)$
          end;
dnini:    $D, U, V := t(f), 1, 1 - q(f)$;
          while $f > 0$ do
dn:         $f, U, V := f{-}1, V, U - q(f{-}1) * V$
          end;
gcd:      $d, u, v := D, U, V$
        end
```

Fig. 2: The extended Euclidian algorithm

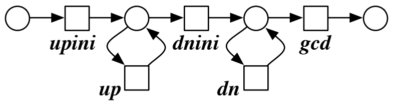


Fig. 3: Collapsed graphical representation of the final gcd model

the use of structured Event-B and small enough to fit fully into this article.

In Section 4.1 we model the program to be developed in models *g0* and *g1*. We refine model *g0* into model *g1* applying Bézout's identity. We introduce the first loop creating a stack of divisions in the second refinement *g2* in Section 4.2, and the second loop in *g3* in Section 4.3. Finally, we data-refine two separate stack pointers used in the two loops into one in *g4* in Section 4.4.

We use the following definitions of *divides*, denoted by |, of *GCD*, and of *abs*:

$$
\begin{array}{lcl}
x|y & \Leftrightarrow & x \neq 0 \wedge (\exists m \cdot y = x * m) \\
z \in GCD[\{x \mapsto y\}] & \Leftrightarrow & z|x \wedge z|y \wedge (\forall d \cdot d|x \wedge d|y \Rightarrow d|z) \\
y = abs(x) & \Leftrightarrow & (x \geq 0 \Rightarrow y = x) \wedge (x < 0 \Rightarrow y = -x)
\end{array}
$$

4.1 GCD by way of a Linear Equation

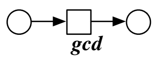
The initial model consists of a single event *gcd*.

```
g0.gcd
  when $b \neq 0$ then
    $d :\in GCD[\{a \mapsto b\}]$
```

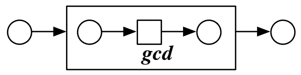
We assume that the variables *a*, *b*, and *d* cannot be data-refined. Similarly to Event-B, we require that variables that are kept in a refinement are implicitly linked by an equality (in all assertions). The initial structure term only states that *gcd* establishes *true* starting from *true* in one step,

$$true \rightarrow \textbf{\textit{g0.gcd}} \rightarrow true$$

There is nothing to prove. Figure 4a shows a graphical representation of the initial model.



(a) Initial gcd model



(b) Refined gcd model

Fig. 4: The first two models of the development of the algorithm

The first refinement replaces the GCD relation by a linear diophantine equation with coefficients u and v:

```
g1.gcd
  when $b \neq 0$ then
    $d,\, u,\, v \;:|\; d' = a * u' + b * v' \;\wedge\; d'|a \;\wedge\; d'|b$
```

The structure of the refinement is the same as that of the abstraction

$$\boldsymbol{g0.gcd} \;\sim\; \mathit{true} \rightarrow \boldsymbol{g1.gcd}! \rightarrow \mathit{true}$$

In the graphical representation we expand the square for event ***g0.gcd*** into a box containing the graphical representation of $\mathit{true} \rightarrow \boldsymbol{g1.gcd}! \rightarrow \mathit{true}$, see Figure 4b. The action simulation proof obligation of the two events ***g0.gcd*** and ***g1.gcd***, $d' = a * u' + b * v' \wedge d'|a \wedge d'|b \Rightarrow d' \in GCD[\{a \mapsto b\}]$ (Bézout's identity), for assertion preservation is easily discharged.

4.2 Creation of a Stack of Divisions

In the second refinement we build up a stack of divisions. Variable h points to the top of the stack that is described by assertion *upinv*:

$$\begin{array}{ll}
@\mathit{upinv} & s \in 0\,..\,h \rightarrow \mathbb{Z} \;\wedge\; t \in 0\,..\,h \rightarrow \mathbb{Z} \\
@\mathit{upinv} & q \in 0\,..\,h \rightarrow \mathbb{Z} \;\wedge\; r \in 0\,..\,h \rightarrow \mathbb{Z} \\
@\mathit{upinv} & h \geq 0 \;\wedge\; s(0) = a \;\wedge\; t(0) = b \\
@\mathit{upinv} & \forall i \cdot i \in 0\,..\,h \;\Rightarrow\; t(i) \neq 0 \;\wedge\; s(i) = t(i) * q(i) + r(i) \\
@\mathit{upinv} & \forall i \cdot i \in 1\,..\,h \;\Rightarrow\; t(i-1) = s(i) \;\wedge\; r(i-1) = t(i)
\end{array}$$

Two new events are introduced. Event ***g2.upini*** initialises the loop that computes the stack and event ***g2.up*** models the loop body.

```
g2.upini
  when $b \neq 0$ then
    $h := 0$
    $s := \{0 \mapsto a\}$
    $t := \{0 \mapsto b\}$
    $q := \{0 \mapsto a \div b\}$
    $r := \{0 \mapsto a \bmod b\}$
```

```
g2.up
  when $r(h) \neq 0$ then
    $h := h+1$
    $s(h+1) := t(h)$
    $t(h+1) := r(h)$
    $q(h+1) := t(h) \div r(h)$
    $r(h+1) := t(h) \bmod r(h)$
```

In the refined event ***g2.gcd*** only the guard is strengthened. The action is unchanged. In fact, the result of the computation is ignored except for the termination condition $r(h) = 0$.

> ***g2.gcd***
> when $r(h) = 0$ then
> $d, u, v :\mid d' = a * u' + b * v' \wedge d'|a \wedge d'|b$

The introduction of the loop is expressed by the term

$$\boldsymbol{g1.gcd} \sim \mathit{true} \rightarrow \boldsymbol{g2.upini} \rightarrow \mathit{upinv} \rightarrow (\boldsymbol{g2.up} \leftarrow [\!] \ \boldsymbol{g2.gcd}! \rightarrow \mathit{true})$$

Figure 5 shows the graphical representation of the model; the square for event

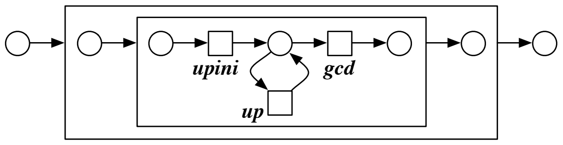


Fig. 5: Second refinement of the gcd model

g1.gcd is replaced by a box representing the refined term. We believe that already this simple case demonstrates the value of the graphical representation. The picture is quite easy to comprehend.

Aside. We are not using the graphical representation to specify structure though: the textual representation is richer and feeding all information into the graphical representation would complicate it. The purpose of the graphical representation is to visualise an important aspect of the model. The syntax of structure terms is designed to resemble the graphical representation.

Assertion preservation is easily proved, for example, for preservation of assertion $\forall i \cdot i \in 0 \,..\, h \Rightarrow t(i) \neq 0$ by events ***g2.upini*** and ***g2.up***,

$$b \neq 0 \Rightarrow \forall i \cdot i \in 0 \,..\, 0 \Rightarrow \{0 \mapsto b\}(i) \neq 0$$
$$\mathit{upinv} \wedge r(h) \neq 0 \Rightarrow \forall i \cdot i \in 0 \,..\, h{+}1 \Rightarrow t \lhd\!\!- \{h{+}1 \mapsto r(h)\}(i) \neq 0$$

Convergence and Enabledness. In the term refining event ***g1.gcd*** we have indicated that event ***g2.up*** terminates. The ring of $\mathbb{Z}$ is a Euclidian domain with abs as a Euclidian function: $\forall x, y \cdot y \neq 0 \Rightarrow (\exists q, r \cdot x = y * q + r \wedge abs(r) < abs(y))$. Hence, the expression $abs(r(h))$ is a variant for event ***g2.up***, that is, $\boldsymbol{g2.up} \leftarrow abs(r(h))$.

The proof obligations for enabledness (showing weakening of the precondition) are $b \neq 0 \Rightarrow b \neq 0$ and $\mathit{upinv} \Rightarrow r(h) \neq 0 \vee r(h) = 0$. Both are easily discharged.

4.3 Calculation of the Coefficients

In the third refinement we calculate the Bézout coefficients u and v and the gcd d by means of the variables D, U, and V. This refinement step is structurally very similar to the second one, except that the stack pointer is decreased during the calculation.

$@dninv\ \ upinv$
$@dninv\ \ k \in 0\,..\,h \ \wedge\ r(h) = 0 \ \wedge\ D|s(k) \ \wedge\ D|t(k)$
$@dninv\ \ D = s(k) * U + t(k) * V$

The proof of the structure refinement

$$\boldsymbol{g2.gcd} \ \sim\ upinv \rightarrow \boldsymbol{g3.dnini} \rightarrow dninv \rightarrow (\boldsymbol{g3.dn} \leftarrow [\!]\ \boldsymbol{g3.gcd}! \rightarrow true)$$

makes use of the properties of the stack described by $upinv$ and requires some arithmetic.

```
g3.dnini                      g3.dn                          g3.gcd
  when $r(h) = 0$ then          when $k > 0$ then              when $k = 0$ then
    $k, D := h, t(h)$             $k := k-1$                     $d := D$
    $U := 1$                      $U := V$                       $u := U$
    $V := 1 - q(h)$               $V := U - q(k-1) * V$          $v := V$
```

In this step we also refine the ***gcd*** event to use the result of the preceding computation of the coefficients and the gcd. Figure 6 shows the graphical representation illustrating the control flow of the algorithm consisting of two consecutive loops preceded by an initialisation each.

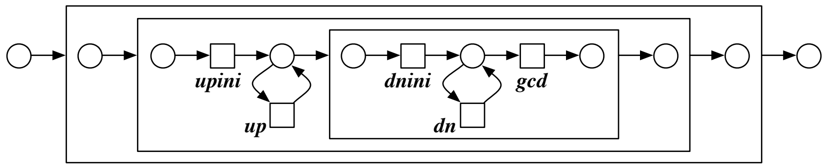


Fig. 6: Third refinement of the gcd model

Convergence and Enabledness. Convergence can be verified by means of the decreasing stack pointer k: $\boldsymbol{g3.dn} \leftarrow k$. The enabledness proof obligations are $upinv \wedge r(h) = 0 \ \Rightarrow\ r(h) = 0$ and $dninv \ \Rightarrow\ k > 0 \vee k = 0$, both proved easily.

4.4 Implementation of the Stack Pointer

We sketch the fourth refinement in order to demonstrate the use of data refinement. In all events of model ***g3*** we textually replace h and k by f. (In refined event ***g4.dnini*** we can remove the resulting assignment $f := f$.)

$$\begin{array}{lll}
\boldsymbol{g2.upini} & \sim & \mathit{true} \rightarrow \boldsymbol{g4.upini}! \rightarrow \mathit{upinv} \\
\boldsymbol{g2.up} & \sim & \mathit{upinv} \rightarrow \boldsymbol{g4.up}! \rightarrow \mathit{upinv} \\
\boldsymbol{g3.dnini} & \sim & \mathit{upinv} \rightarrow \boldsymbol{g4.dnini}! \rightarrow \mathit{dnini} \\
\boldsymbol{g3.dn} & \sim & \mathit{dninv} \rightarrow \boldsymbol{g4.dn}! \rightarrow \mathit{upinv} \\
\boldsymbol{g3.gcd} & \sim & \mathit{dninv} \rightarrow \boldsymbol{g4.gcd}! \rightarrow \mathit{true}
\end{array}$$

The assertions *upinv* and *dninv* are extended by gluing assertions relating model ***g3*** to model ***g4***

$$\begin{array}{ll}
@\mathit{upinv} & f = h \\
@\mathit{dninv} & f = k
\end{array}$$

Aside. In the algorithm shown in Figure 2 all variables are global. We could as well have inferred from the structure of the model (Figure 6) local variables f, s, t, q, r for the two loops and D, U, V for the second loop.

5 Related and Future Work

The two verification approaches presented in [16] and [14] are lacking a notion of refinement. In [8] a restricted form of refinement for temporal verification diagrams is presented that permits splitting vertices and removing edges. This is generalised in [7] by considering the transitive closure of edges and matching notion of data-refinement. In our approach, we have incorporated refinement based on the corresponding notion of Event-B that appears simpler to handle. We also preserve structure information during refinement which is particularly important for obtaining the intended algorithmic structure by the end of a development.

Alternative ways of expressing structure of Event-B models that have been proposed are the CSP-based approach of [13], JSD-based approach of [6] and UML-B [17]. In [13] events are annotated with events that are to be enabled *next*. Corresponding enabledness proof obligations are shown but refinement is not considered. In [6] JSD-like diagrams are used to illustrate concurrent Event-B models. The notation can also be used to illustrate refinement of Event-B models. However, the notation is not exploited for proof obligation generation; the suggested (still) informal mapping to Event-B introduces abstract program counters. UML-B [17] focuses more on states as its central concept. UML-B models are translated into Event-B by introducing abstract program counters to represent those states. The notion of refinement is centred around state decomposition and gluing invariants are generated from the emerging nesting structure of state machines. State machines of UML-B are not exploited for proof obligation generation.

Abstract State Machines (ASM) [5] also provide to ways to introduce structure into a model. Control State ASMs use abstract program counters to model control structures. We could also identify such a class of models in Event-B. But this would not solve our problem. One of the reasons for introducing a structure notation in Event-B is that proof obligations involving program counters can get quite involved. Apart from that invariants of Event-B get cluttered with properties involving abstract program counters. A second way to structure Abstract State Machines is to use Turbo ASMs [5]. Turbo ASMs provide programming constructs to compose ASMs to model computations. Such a reconstruction of programming constructs would not solve our problem either. We would again have to deal with sequential composition, if-statements, and so on, when generating proof obligations.

We are investigating modelling concurrency ($p_1 \to e_1 \to q_1 \parallel p_2 \to e_2 \to q_2$) using structured Event-B. In fact, the proof outlines in [16] were first developed for proving properties of concurrent programs. We are more interested to look into possibilities for support by a tool such as the Rodin tool [2]. The proof obligations for enabledness are difficult to handle. However, this difficulty is also present in unstructured Event-B models: in structured Event-B it will surface during proof obligation generation, whereas in unstructured Event-B it will show up during proof.

We also think about changes concerning the way proof obligations are generated. In the new scheme of proof obligation generation we would no longer get a list of all proof obligations that must be discharged. Instead, we would get a todo-list that tells us what still needs to be proved. We have realised that this is the way we should proceed with action feasibility and convergence proof obligations. Action feasibility can be proved showing the existence of a post-state directly or by implementing the action in a refinement. Convergence proofs can be delayed by using anticipation. This approach would make the Event-B method more flexible without sacrificing its strong tool support.

We are also investigating verification of temporal properties. The temporal verification diagrams in [14] are used to prove temporal logic properties of reactive systems. A similar approach should also work for structured Event-B. Refinement should provide a way to master more complex temporal properties. The structured Event-B approach shares with temporal verification diagrams the strength of only generating first-order proof obligations.

6 Conclusion

We have introduced a structure notation for Event-B together with the necessary proof obligations. We have demonstrated how it can be used practically in a sequential program development. The structure notation is *not* a programming notation but a notation that describes a theory about a formal model. For instance, the formula $p \to e \to q \to f \to r$ does not mean: first e is executed then f; it describes theorems about e and f. Effectively, we have moved some concepts of proof into modelling. We think this is very attractive, in particular, for implementation in a software tool such as Rodin. There would be no need to configure the proof obligation generator for different applications. Everything about the proof obligations would be said in the model; fully transparent for the user of the tool. An additional benefit would be that only proof obligations need to be generated that are specified in structure terms. Hence, usually, fewer proof obligations would be generated.

In our opinion, the notation also improves legibility of more complex structured models. The associated graphical notation helps to grasp quickly the structure of a model.

Acknowledgement. The original (unstructured) Event-B model of the extended Euclidian algorithm was developed by Christophe Métayer. Jens Bendisposto provided useful comments on earlier versions of this article.

References

1. J.-R. Abrial. *Modeling in Event-B: System and Software Engineering.* Cambridge University Press, 2009. To appear.
2. J.-R. Abrial, M. Butler, S. Hallerstede, and L. Voisin. An open extensible tool environment for Event-B. In Z. Liu and J. He, editors, *ICFEM 2006*, volume 4260, pages 588–605. Springer, 2006.
3. J.-R. Abrial and S. Hallerstede. Refinement, Decomposition and Instantiation of Discrete Models: Application to Event-B. *Fundam. Inform*, 77(1-2):1–28, 2007.
4. K. R. Apt, , F. S. de Boer, and E.-R. Olderog. *Verification of Sequential and Concurrent Programs.* Springer-Verlag, 2009.
5. E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis.* Springer-Verlag, 2003.
6. M. Butler. Decomposition Structures for Event-B. In M. Leuschel and H. Wehrheim, editors, *IFM*, volume 5423 of *LNCS*, pages 20–38. Springer, 2009.
7. D. Cansell, D. Méry, and S. Merz. Diagram refinements for the design of reactive systems. *J. UCS*, 7(2):159–174, 2001.
8. L. de Alfaro, Z. Manna, H. B. Sipma, and T. E. Uribe. Visual verification of reactive systems. In E. Brinksma, editor, *TACAS*, volume 1217 of *LNCS*, pages 334–350. Springer, 1997.
9. S. Hallerstede. Justifications for the Event-B Modelling Notation. In J. Julliand and O. Kouchnarenko, editors, *B 2007*, volume 4355 of *LNCS*, pages 49–63. Springer, 2007.
10. S. Hallerstede. On the Purpose of Event-B Proof Obligations. In E. Börger, M. J. Butler, J. P. Bowen, and P. Boca, editors, *ABZ*, volume 5238 of *LNCS*, pages 125–138. Springer, 2008.
11. S. Hallerstede. Proving Quicksort Correct in Event-B. Refine 2009. 2009. 16 pages.
12. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 583, 1969.
13. W. Ifill, S. A. Schneider, and H. Treharne. Augmenting B with control annotations. In J. Julliand and O. Kouchnarenko, editors, *B 2007*, volume 4355 of *LNCS*, pages 34–48. Springer, 2007.
14. Z. Manna and A. Pnueli. Temporal verification diagrams. In M. Hagiya and J. C. Mitchell, editors, *Theoretical Aspects of Computer Software*, volume 789 of *LNCS*, pages 726–765. Springer, 1994.
15. C. Morgan. *Programming from Specifications: Second Edition.* Prentice Hall, 1994.
16. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6(4):319–340, 1976.
17. M. Y. Said, M. J. Butler, and C. F. Snook. Language and tool support for class and state machine refinement in UML-B. In A. Cavalcanti and D. Dams, editors, *FM 2009*, volume 5850 of *LNCS*, pages 579–595. Springer, 2009.