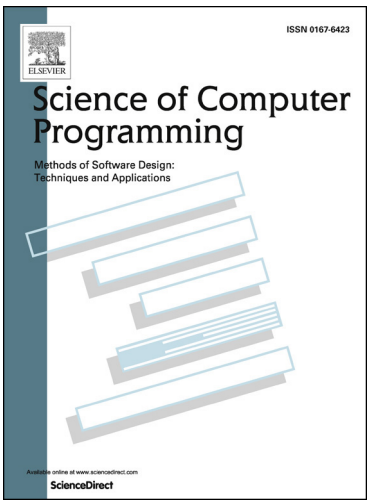
Accepted Manuscript



Derivation of algorithmic control structures in Event-B refinement

Mohammadsadegh Dalvandi, Michael Butler, Abdolbaghi Rezazadeh

PII: S0167-6423(17)30120-X
DOI: http://dx.doi.org/10.1016/j.scico.2017.05.010
Reference: SCICO 2103

To appear in: *Science of Computer Programming*

Received date: 27 May 2016
Revised date: 30 May 2017
Accepted date: 31 May 2017

Please cite this article in press as: M. Dalvandi et al., Derivation of algorithmic control structures in Event-B refinement, *Sci. Comput. Program.* (2017), http://dx.doi.org/10.1016/j.scico.2017.05.010

Highlights

- Introducing an approach for making the control flow in Event-B explicit.
- The approach augments Event-B with a scheduling language for specifying the control flow.
- A number of refinement patterns for refining a schedule to a concrete level are introduced.
- The correctness of the control flow is verified by removing the original guards from events in a schedule.
- The approach is applied to the famous Schorr-Waite algorithm.

Derivation of Algorithmic Control Structures in Event-B Refinement

Mohammadsadegh Dalvandi, Michael Butler, Abdolbaghi Rezazadeh

University of Southampton, Southampton SO17 1BJ

Abstract

The Event-B formalism allows program specifications to be modelled at an abstract level and refined towards a concrete model. However, Event-B lacks explicit control flow structure and ordering is implicitly encoded in event guards. This makes it difficult to identify and apply rules for transformation of Event-B models to sequential code. This paper introduces a scheduling language to support the incremental derivation of algorithmic control structure for events as part of the Event-B refinement process. We provide intermediate control structures for non-deterministic iteration and choice that ease the transition from abstract specifications to sequential implementations. We present rules for transforming algorithmic structures to more concrete refinements. We illustrate our approach by applying our method to the Schorr-Waite graph marking algorithm.

Keywords:
Refinement, Program verification, Event-B, Program derivation

1. Introduction

Correct software can be developed by formally deriving an implementation from its abstract specification in a number of successive refinement steps and proving correctness against a high level specification. Event-B [1] is a formal modelling language that follows this approach. The development of a software system starts with a high level specification and continues with

Email addresses: md5g11@ecs.soton.ac.uk (Mohammadsadegh Dalvandi), mjb@ecs.soton.ac.uk (Michael Butler), ra3@ecs.soton.ac.uk (Abdolbaghi Rezazadeh)

refining the abstract model towards a concrete one. Event-B uses guarded atomic actions for specifying and modelling a system. This is convenient for modelling and localising the refinement reasoning. Event-B supports data refinement and atomicity decomposition through introduction of new events. It has a rich set theoretic mathematical language and strong tool support for verification (Rodin [2]).

Event-B lacks explicit control flow structure and ordering should be implicitly encoded in the event guards. This makes it difficult to identify and apply rules for transformation of Event-B models to sequential code.

In this paper, we introduce a scheduling language to support the incremental derivation of algorithmic control structure for events as part of the Event-B refinement process. We introduce operators for sequential composition, choice and iteration. Our scheduling language supports non-deterministic choice and iteration of event groups for use at the intermediate levels of refinement. This allows us to retain event guards locally for reasoning about data refinement. Event groups allow us to model different cases at both the specification and implementation level. Closer to the implementation level, non-deterministic choices are replaced by deterministic if-statements and non-deterministic iterations by deterministic while-statements. We present a number of refinement rules to assist the modeller in refining the abstract schedule towards the concrete level. To illustrate our method we apply it to the well-known Schorr-Waite graph marking algorithm.

Our approach is influenced by the refinement calculus [3, 4, 5]. The refinement calculus extends the guarded command language [6] of Dijkstra with a *specification statement*. A specification statement comprises two predicates *pre* and *post* over the program variable v. This means that if the initial state satisfies *pre*, the execution of the statement terminates in a state satisfying *post*. A specification in the refinement calculus is comprised of executable statements (called *code*) and specification statements. The refinement calculus provides laws for replacing non-deterministic specification statements with deterministic program structures (sequential composition, if-statements and while-statements). In our approach we introduce non-deterministic choice and iteration before if-statements and while-statements. This allows us to retain guards with events and use the guards to keep the reasoning (including data refinement) localised to events or pairs of corresponding abstract and refined events. This style of reasoning corresponds directly to Event-B refinement and allows us to mix program refinement and data refinement by using an existing tool (Rodin) to represent and prove data

refinements. After introducing the deterministic program structure, the control guards in a schedule provide derived guards for events that allow the elimination of the event guards and thus they may be removed from a final implementation.

The rest of the paper is organised as follows: in Section 2 a brief introduction on Event-B is given. Section 3 explains the abstract scheduling language and refinement, Section 4 introduces the concrete scheduling language, refinement rules, and guard elimination conditions. Section 5 presents the case study and elaborates on the methodology. Section 6 discusses related and future works and finally Section 7 concludes the paper.

2. Background: Event-B Modelling Language

Event-B is a formal modelling language based on set theory and predicate logic for specifying, modelling and reasoning about systems, introduced by Abrial [1]. Event-B is greatly inspired by the notion of Action Systems [7] and Guarded Commands [6]. The structure of an Event-B specification is similar to those in Action Systems where the behaviour of the system is specified by guarded actions.

A model in Event-B consists of two main parts: *contexts* and *machines*. The static part (types and constants) of a model is placed in a context and specified using carrier sets, constants and axioms. The dynamic part (variables and events) is specified in a machine by means of variables, invariants and events. A context, say C, can be seen by one or more machines and it can be extended by another context D. A machine, say M, can be refined by another machine N ($M \sqsubseteq N$).

In a context, carrier sets are represented by their name and they are disjoint from each other. Constants are defined using axioms. Axioms are predicates that express properties of sets and constants. Sets and constants are denoted by s and c respectively. Axioms are represented by $P(s, c)$.

A machine in Event-B consists of three main elements: (1) a set of *variables*, which defines the states of a model (2) a set of *invariants*, which is a set of conditions on state variables that must be preserved before and after execution of any event and (3) a number of *events* which model the state changes in the system. Each event may have a number of assignments called *actions*. Each event may also have a number of *guards*. Guards are predicates that describe the necessary conditions which should be true before an event can occur. An event may have a number of parameters. Event parameters

are considered to be local to the event. Variables and invariants are denoted by v and $I(s, c, v)$, respectively. An event *evt* has the following form:

$$\text{evt} \triangleq \textbf{any } p \textbf{ where } G(p, s, c, v) \textbf{ then } v \coloneqq E(p, s, c, v)$$

where p, $G(p, s, c, v)$, and $v \coloneqq E(p, s, c, v)$ represent event parameters, guards, and actions, respectively. All events should preserve the model invariants. The invariant preservation proof within Event-B guarantees the following for each event *evt*:

$$I(s, c, v) \wedge P(s, c) \wedge G(p, s, c, v) \wedge BA(p, c, v, v') \Rightarrow I(s, c, v')$$

where $BA(p, c, v, v')$ represents the *before-after predicate* for *evt*. The before-after predicate denotes the relationship between the value of variables just before and just after the execution of the event. By convention, primed variables (v') refer to the value of the variables after the execution of the event.

In an Event-B model if all of the events are disabled (i.e. guards are false) then a deadlock has happened. In some models this might be fine. However, if a deadlock is not desirable in a model then a proof obligation called *deadlock freedom* should be proved. This proof obligation states that at least one of the guards of the events of the model is always true. If model axioms are denoted by $A(s, c)$, for a model with n events, this proof obligation guarantees the following:

$$A(s, c) \wedge I(s, c, v) \Rightarrow G_1(p, s, c, v) \vee ... \vee G_n(p, s, c, v)$$

Axioms, invariants, and guards can be marked as theorems. This means that the validity of the theorem should be proved by axioms, invariants, and guards which appeared before the theorem [8]. To prove that events *terminate* and a chosen set of events are enabled only a finite number of time before disabling themselves they may be labelled as *convergent*. A convergent event requires a *variant*. A variant is a natural number expression which must be decreased by all *convergent* events. An event can be labelled as *anticipated*. This allows the modeller to postpone the proof of termination to a later refinement.

Modelling a complex system in Event-B largely benefits from refinement as modelling a system at different levels of abstraction helps to tackle the complexity of design. Refinement is a stepwise process of building a large

system starting from an abstract level towards a concrete level. This is done by a series of successive steps in which, new details of functionality are added to the model in each step. The abstract level represents key features and the main purpose of the system. The abstract model does not include implementation details and *how* the goal of the system is going to be achieved. Instead, it focuses on *what* is the goal of the system.

In each refinement all abstract events are refined by one or more concrete events. Each refinement may involve introducing new variables to the model. This usually results in extending abstract events or adding new events to the model. All new events refine a dummy event at the abstract machine. This dummy event does nothing. If an abstract data type is replaced with a concrete one in the refined level, a *gluing invariant* is used to relate the abstract and concrete states. A gluing invariant refers to both abstract and concrete variables. A new event may be marked as *convergent*. If the refined machine contains convergent events then a *variant* should be defined. A variant is either a natural number expression which is decreased by the convergent events or a finite set expression which is made smaller by the convergent events. Convergence of the variant ensures that execution of new events will terminate. A new event may be marked as anticipated if it is expected to converge but proof of convergence is postponed to later refinement steps. Convergence is not compulsory and depends on the purpose of the new events. If the new event is used to represent atomicity decomposition, then convergence is required.

Abstract variables can be replaced by newly defined concrete variables (i.e. data refinement). Event-B events can be treated as guarded actions from [9, 10]. Data refinement of guarded actions in [9, 10] corresponds to data refinement of events in Event-B. We use $\sqsubseteq_R$ to denote data refinement through relation R:

$$R = \{a \mapsto c \mid I(a, c)\}$$

where a and c represent the list of abstract and concrete variables, respectively, and $I(a, c)$ represents the gluing invariant in the Event-B model. This is done by treating events as relations on before-after state [1], i.e., an abstract event relates a and a' while a concrete event relates c and c'. Assume S and T are the relational representations of events, then $S \sqsubseteq_R T$ is defined as:

$$T \wedge I \;\Rightarrow\; \exists\, a.S \wedge I[a, c := a', c']$$

The refinement proof obligations of Event-B are derived from this definition [1] and are implemented in the Rodin tool [2].

3. Abstract Algorithmic Refinement with Event-B

A typical Event-B development starts with an abstract model of the system with a small number of events specifying the ultimate goal of the system. The development continues by refining the abstract model and adding new variables and events in order to model the functional behaviour of the system. In each refinement level, the model is nothing more than a number of variables, invariants and events and possibly a context which is seen by the model. Each event can be executed only when it is enabled (i.e. its guards are true). If there is more than one enabled event, then the next event to be executed is chosen non-deterministically. If a strict control flow between events is required, then it should be encoded in the event guards. This non-determinism and implicit control flow has several shortcomings for developing verified programs with Event-B. First, it makes the program derivation specifically for sequential programs difficult (because of the implicit control flow). Second, making the decision about the program structure is left for the concrete level which might not be easy. Third, for a programmer who wants to employ a formal approach like Event-B, it may be easier to have a code-like structure expressing the algorithmic structure of each refinement and its relation to the abstract level. This code-like (program constructs) structure is missing in Event-B.

To overcome the above shortcomings, we have augmented Event-B models with an explicit scheduling language and provided a number of rules for helping the modeller to refine the abstract program structure to a concrete level. This section defines our abstract scheduling language and abstract algorithmic refinement rules. It motivates the refinement rules through a simple binary search algorithm. In the future sections we will present our concrete scheduling language and validate our approach by applying it to a larger case study (the Schorr-Waite algorithm).

3.1. Abstract Scheduling Language

We augmented Event-B with a scheduling language to make the structured program associated with each level of refinement explicit. In our approach, in addition to the standard Event-B machine components (i.e. variables, invariants, variants, and events), each model has an associated

schedule. A schedule allows us to define a structured program with guarded events as its atoms. A schedule associated with an abstract model comprises non-deterministic control constructs since the model events usually have non-deterministic actions and/or abstract data types. The syntax of our proposed abstract scheduling language is shown using Backus-Naur Form (BNF) in Figure 1.

$$
\begin{array}{lcl}
\langle \mathit{Schedule} \rangle & ::= & \mathit{Event} \\
& | & \langle \mathit{Schedule} \rangle \ ; \ \langle \mathit{Schedule} \rangle \\
& | & \langle \mathit{Schedule} \rangle \ \Box \ \langle \mathit{Schedule} \rangle \\
& | & \langle \mathit{Schedule} \rangle^*
\end{array}
$$

Figure 1: The Abstract Scheduling Language

The simplest form of a schedule is a single event. *Event* denotes the label of an event in a machine. A schedule may contain one or more event labels. A sequential order can be imposed by the sequential composition operator (;). Non-deterministic choice ($S_1 \ \Box \ S_2$) and iteration (S^*) are abstract control structures. Iteration is required to be finite. This is enforced by proving convergence of the events in an iteration. The aforementioned control structures allow us to retain the event structure (guards and actions together) so that the reasoning about data refinement is localised to pairs of corresponding abstract and refining events using the standard definition of the Event-B refinement.

3.1.1. Example: A Binary Search Algorithm

Here we present a simple example (a binary search algorithm) to illustrate how a sequential program can be developed in Event-B and how we can make the control flow explicit using the abstract scheduling language introduced in the previous section.

The algorithm was originally modelled in [1] with minor differences. Assume that we have a sorted array of integers f and a target value v where v exists in f. The goal of the algorithm is to find the position of v in f. If the size of the array is n, the context of our model will have the following axioms:

axm1: $f \in 0..n\text{-}1 \rightarrow \mathbb{Z}$

axm2: $v \in ran(f)$
axm3: $\forall x, y.\, x \in 0..n-1 \wedge y \in 0..n-1 \wedge x \leqslant y \;\Rightarrow\; f(x) \leqslant f(y)$

The array is specified using a total function ($axm1$), v is in the array ($axm2$) and the array is sorted ($axm3$). These axioms can be considered as pre-conditions of our algorithm. The abstract specification is very simple. It has only one event (*found*) apart from the initialisation event. We use a result variable r to specify the desired outcome of the algorithm:

Machine $m0$ **Sees** $c0$
Variables r
Invariants $r \in 0..n-1$
Initialisation $r := 0$

Event $found$
any e
where
grd1: $f(e) = v$
then
act1: $r := e$
End

The *found* event finds the position of v in *one shot* and specifies *what* should be achieved by the algorithm which is yet to be developed. One can capture the control flow between events of the model using the proposed scheduling language:

$$initialisation\,;\, found$$

This means that the execution of the initialisation event followed by the *found* achieves the goal of the algorithm. There is no detail in the abstract specification about how the goal is going to be achieved. The future refinement steps will address this. In the next refinement we need a new event *search* in order to find the position of v in the array. We specify that the execution of event *search* for a finite number of iterations should result in finding the position of v in the array. This can be captured by the scheduling language as follows:

$$initialisation\,;\, search^{*}\,;\, found$$

The associated Event-B model with the above schedule is as follows:

Machine $m1$ **refines** $m0$ **Sees** $c0$
Variables r, k
Invariants $k \in 0..n-1$
Initialisation $r := 0, k :\in 0..n-1$

Event *search*
where
grd1: $f(k) \neq v$
then
act1: $k :\in 0..n-1$
End

Event $found$
refines $found$
where
grd1: $f(k) = v$
then
act1: $r := k$
End

where k is a new variable used in the computation to find the position of v in f. The *search* event is responsible for finding the index of v in f. Each execution of the *search* event assigns an integer between 0 and $n-1$ to k non-deterministically. Once *search* finds a value for k that satisfies $f(k) = v$, then *search* is disabled and $found$ is enabled. Note that at this stage *search* is an anticipated event whose convergence is proved in the next refinement. The above schedule is a refinement of the abstract schedule just like the events in the schedule that refine the abstract ones. We use the $\sqsubseteq_R$ symbol to indicate the schedule refinement and $S_0 \sqsubseteq_R S_1$ is read as "*S_0 is refined by S_1*" where R represents the data refinement relation defined by the invariants of the refined machine:

$$\begin{array}{c} initialisation\,;found \\ \sqsubseteq_R \\ initialisation\,;search^*\,;found \end{array}$$

The above schedule specifies that before the final goal of the algorithm is achieved, event *search* may be executed a number of times. This indicates the existence of a loop in the final program. Although now we have specified details of *how* the goal is going to be achieved, still the model contains non-deterministic assignments and does not implement a binary search algorithm. The next refinement replaces this with a binary search. The schedule refinement for this level is as follows:

$$\begin{array}{c} initialisation\,;search^*\,;found \\ \sqsubseteq_R \\ initialisation\,;(search\_inc \,\Box\, search\_dec)^*\,;found \end{array}$$

In this refinement two new variables i and j are defined such that the index of v is between i and j ($v \in f[i..j]$, specified in the invariant). The

search event is refined by two separate events, *search_inc* and *search_dec* which bring i and j closer together. Events *search_inc* and *search_dec* are marked as convergent and we introduce a variant $(j - i)$ to prove the termination:

Machine $m2$ **refines** $m1$ **Sees** $c0$
Variables r, k, i, j
Invariants $i \in 0..n-1 \wedge j \in 0..n-1 \wedge k \in i..j \wedge v \in f[i..j]$
Variant $j - i$
Initialisation $r := 0, k :\in 0..n-1, i := 0, j := n-1$

Event *search_inc*
refines *search*
where
grd1: $f(k) < v$
then
act1: $k :\in k+1..j$
act2: $i := k+1$
End

Event *search_dec*
refines *search*
where
grd1: $f(k) > v$
then
act1: $k :\in i..k-1$
act2: $j := k-1$
End

Event $found$
refines $found$
where
grd1: $f(k) = v$
then
act1: $r := k$
End

The model still has non-deterministic assignments (the value of variable k is assigned non-deterministically). The next refinement will replace the non-deterministic actions with deterministic ones:

Machine $m3$ **refines** $m2$ **Sees** $c0$
Variables r, k, i, j
Initialisation $r := 0, k := (n-1)/2, i := 2, j = n-1$

Event *search_inc*
refines *search*
where
grd1: $f(k) < v$
then
act1: $k := (k+j+1)/2$
act2: $i := k+1$
End

Event *search_dec*
refines *search*
where
grd1: $f(k) > v$
then
act1: $k := (i+k-1)/2$
act2: $j := k-1$
End

Event $found$
refines $found$
where
grd1: $f(k) = v$
then
act1: $r := k$
End

This refinement adds no new event to the model and does not change

its algorithmic structure so the associated schedule remains the same. It is worth noting that so far this derivation involved abstract program refinement (introduction of *search*) prior to data refinement (introduction of i and j). However there is still a gap between the non-deterministic schedule that was introduced in the last refinement and the concrete algorithmic structure containing `if` and `while` rather than $\Box$ and $*$ needed for the final program code. Before we introduce the concrete schedule language and refinement, first we formalise the introduction and refinement of the abstract schedules in the next section.

3.2. Abstract Schedule Refinement

Event-B refinement is performed at event granularity where each event of a refinement either refines some abstract event or refines `skip`. Correctness of the Event-B refinement is verified through a number of proof obligations. Using Event-B and our scheduling language we can develop structured programs in a stepwise refinement. To accomplish this, here we introduce a number of new rules for *abstract schedule refinement*. Abstract schedule refinement is about elaborating the schedule in tandem with event refinement and the introduction of new events.

Before we present the schedule refinement rules, we need to set a few conventions and rules. First, we use lower case letters to represent individual events and upper case letters to represent *event groups*. An event group is a non-deterministic choice between a list of n events:

$$E = e_1 \Box e_2 \Box ... \Box e_{\mathrm{n}}$$

Second, if event group E is refined by event group E' of the same length, written $E \sqsubseteq_R E'$, then there is a pairwise refinement relation between events in E and E' as follows:

$$e_1 \sqsubseteq_R e'_1, e_2 \sqsubseteq_R e'_2, ..., e_{\mathrm{n}} \sqsubseteq_R e'_n$$

Third, from the literature [3, 11] we know that the sequential composition ($;$), non-deterministic choice ($\Box$), and iteration ($*$) operators are monotonic with respect to refinement, i.e., if $S \sqsubseteq_R S'$ and $T \sqsubseteq_R T'$ then:

$$S ; T \sqsubseteq_R S' ; T'$$
$$S \Box T \sqsubseteq_R S' \Box T'$$
$$S^* \sqsubseteq_R (S')^*$$

Finally, to avoid confusion, it is worth mentioning that syntactically a schedule uses event labels. However, semantically, the schedule and the definitions of the events define a structured program. The refinement rules are justified at the semantic level.

We introduce the following general rules for algorithmic and schedule refinement:

- **Rule 1 (Case Split)**

  $e \sqsubseteq_R (e'_1 \,\square\, ... \,\square\, e'_m)$ (provided $e \sqsubseteq_R e'_1$,..., $e \sqsubseteq_R e'_m$)

  In a typical Event-B refinement, an abstract event may be refined by one or more events at the concrete level. If an event is refined by more than one event, each of the refined events is typically used to represent a different case of the abstract event. An abstract schedule with one event (e) can be replaced with a concrete schedule with a group of m events ($e'_1 \,\square\, ... \,\square\, e'_m$).

  Proof. The case split refinement rule immediately follows from the monotonicity property of the non-deterministic choice operator. □

  Remember deadlock freedom proof obligation from Section 2. Here we also need to prove that the enabledness condition, that generalises the deadlock freeness, is preserved by the case events. This means that whenever the abstract event (e) is enabled at least one of the concrete events ($e'_1, ..., e'_m$) should be enabled:

  $$grd(e) \;\Rightarrow\; grd(e'_1) \vee ... \vee grd(e'_m)$$

  where $grd(e)$ represents the guards of event e. We will show in Section 4.3 that if the enabledness condition is not satisfied then we would not be able to eliminate event guards in the final program.

- **Rule 2 (Loop Introduction)**

  $E \sqsubseteq_R F^* \,;E'$ (provided $\texttt{skip} \sqsubseteq_R F$ and $E \sqsubseteq_R E'$)

  Here the abstract schedule E is replaced by the schedule $F*;E'$ which specifies that some event from F is continually executed until an event in E' is enabled. In the later refinements F^* may be be refined by a concrete loop.

Proof. This refinement rule can be proved as follows:

$$\begin{aligned} E &\sqsubseteq \texttt{skip} \,;\, E \\ &\sqsubseteq \texttt{skip}^* \,;\, E \\ &\sqsubseteq_R \{\textit{monotocity of * and ; and * represents finite iteration}\} \\ &F^* \,;\, E' \end{aligned}$$

□

Since we have to prove loop termination, F should be either labelled as *convergent* or *anticipated*. If it is labelled as *convergent* then a variant should be provided in order to prove the termination of each event in F. The proof of termination can be postponed for any of the F events until future refinements by marking them as *anticipated*. Proving the convergence ensures the termination of the loop.

Similar to the previous rule, the enabledness condition should be preserved:

$$grd(E) \;\Rightarrow\; grd(F) \vee grd(E')$$

If E is an event group then $grd(E)$ represents the disjunction of the guards of the events in E.

- **Rule 3 (Loop Body Extension)**

  $E^* \sqsubseteq_R (F \,\square\, E')^*$ (provided $\texttt{skip} \sqsubseteq_R F$ and $E \sqsubseteq_R E'$)

  The body of an abstract loop may be extended by an event group in a refinement.

  Proof. The loop body extension rule can be proved by induction:

$$\begin{aligned} E^* &\sqsubseteq (\texttt{skip} \,\square\, E)^* \\ &\sqsubseteq_R \{\textit{monotonicity of } \square \textit{ and } *\} \\ &(F \,\square\, E')^* \end{aligned}$$

  □

  Also like the previous rule, F should be labelled as *convergent* or *anticipated* and the enabledness condition should be proved.

Nested control structures can be derived by repeated application of these refinement rules.

4. Refinement to Concrete Program Structures

The scheduling language and refinement rules introduced in the previous section provide a way to specify the control flow between events and to refine it. However, due to the fact that the schedule itself contains non-deterministic choices and iterations there is still a gap between the abstract and the concrete algorithmic structure of the final program code. To bridge the gap we extend the abstract scheduling language introduced in Section 3.1 with deterministic branches and loops. The language is presented in EBNF [12] in Figure 2.

⟨*Schedule*⟩ ::= *Event*
| ⟨*Schedule*⟩ ; ⟨*Schedule*⟩
| ⟨*Schedule*⟩ □ ⟨*Schedule*⟩
| ⟨*Schedule*⟩*
| **if**(⟨*Cond*⟩){⟨*Schedule*⟩}, {**elseif**(⟨*Cond*⟩){⟨*Schedule*⟩}},[**else**{⟨*Schedule*⟩}]
| **while**(⟨*Cond*⟩){⟨*Schedule*⟩}

⟨*Cond*⟩ ::= *Predicate*

Figure 2: The Scheduling Language

The extension adds deterministic `if..else` branches and `while` loops with explicit conditions (*Cond*) to the language. The branch and loop conditions should be valid Event-B predicates as defined in [1]. Non-deterministic choices and iterations can be refined to deterministic branches and loops, respectively. Deterministic branches and loops can be defined in terms of non-deterministic choices and iterations and guards [3]:

$$\mathbf{while}(cond)\{S\} \triangleq ([cond]\,;S)^{\omega}\,;\,[\neg cond]$$
$$\mathbf{if}(cond)\{S_1\}\mathbf{else}\{S_2\} \triangleq [cond]\,;S_1 \,\square\, [\neg cond]\,;S_2$$

Guards (or assumptions) are conditions on the state of the program at the point which they occur. A guard is supposed to hold at the point which it is defined and does not change the state. A guard is expressed by enclosing the condition in brackets (e.g. $[cond]$). The while-loop is defined using the *strong iteration* operator. Strong iteration S^{ω} means that S may be executed finite or infinite number of times. Weak iteration S^{*}, on the other hand, means

that S may only be executed a finite number of times. Since we ensure termination by proving convergence we can replace strong iteration (S^ω) with weak iteration (S^*) so we have the following definition for a while-loop:

$$\textbf{while}(cond)\{S\} \triangleq ([cond]\,;S)^*\,;\,[\neg cond]$$

In order to be able to keep data and algorithmic refinements separate we postpone the refinement of abstract control structures (i.e. choice and iteration) to concrete structures until the data refinement is completed. The following two schedule refinement rules will allow us to refine an abstract schedule with non-deterministic control structures to a concrete control structure:

- **Rule 4 (Concrete Loop)** $S^* \sqsubseteq \textbf{while}(cond)\{S\}$

  An abstract loop can be refined by a deterministic while-loop. The modeller should explicitly determine the concrete loop condition ($cond$).

  Proof. The refinement of an abstract iteration to a concrete while-loop can be proved as follows:

  $$\begin{aligned} & S \sqsubseteq \texttt{skip}\,;\,S \\ & \quad \sqsubseteq \{\text{guards refine } \texttt{skip}\} \\ & [cond]\,;S \end{aligned}$$

  Since * is monotonic:

  $$\begin{aligned} S^* & \sqsubseteq ([cond]\,;S)^* \\ & \sqsubseteq ([cond]\,;S)^*\,;\,\texttt{skip} \\ & \sqsubseteq \{\text{guards refine } \texttt{skip}\} \\ & ([cond]\,;S)^*\,;[\neg cond] \\ & = \{\text{definition of while-loop}\} \\ & \textbf{while}(cond)\{S\} \end{aligned}$$

  □

- **Rule 5 (Concrete Branch)**

  $S_0 \,\square\, S_1 \,\square\, ... \,\square\, S_m$

  $\sqsubseteq$

if(*cond1*)**{**S_1 **}elseif**(*cond2*)**{**S_2**}**...**else{**S_m**}**

Non-deterministic choice between two or more schedules can be refined to an *if..else* structure. The modeller should explicitly provide branch conditions.

Proof. Here, for simplification, we present a proof for a simple case when there are only two event groups involved in the non-deterministic choice:

$$
\begin{aligned}
S_1 \,\square\, S_2 &\sqsubseteq (\texttt{skip}\,;\, S_1) \,\square\, (\texttt{skip}\,;\, S_2) \\
&\sqsubseteq \{\text{guards refine } \texttt{skip} \text{ and monotonicity of } \square\} \\
&([cond]\,;\, S_1) \,\square\, ([\neg cond]\,;\, S_2) \\
&= \{\text{definition of branch}\} \\
&\textbf{if}(cond)\textbf{\{}S_1\textbf{\}else\{}S_2\textbf{\}}
\end{aligned}
$$

□

The proof is easily generalised to the case of m branches.

In Section 4.2, we will discuss how we can use the explicit schedule conditions to eliminate event guards.

4.1. Refining the Abstract Schedule of the Search Example

Remember the example from 3.1. We refined the abstract specification to a level where the model was completely deterministic and no further data refinement was required. Now that our scheduling language has deterministic loops and branches, we can refine its non-deterministic schedule to a deterministic one. This is done in three steps. We first refine the loop body to a deterministic branch:

$$
\begin{gathered}
search\_inc \,\square\, search\_dec \\
\sqsubseteq \\
\textbf{if}(f(k)<v)\textbf{\{}search\_inc\textbf{\}}\ \textbf{else\{}search\_dec\textbf{\}}
\end{gathered}
$$

The body of the loop is replaced by its refinement and the next step is to replace the non-deterministic iteration with a deterministic while-loop:

$$initialisation\,;\,(\textbf{if}(f(k)<v)\{search\_inc\}\ \textbf{else}\{search\_dec\})^*\,;\,found$$
$$\sqsubseteq$$
$$initialisation;$$
$$\textbf{while}(f(k)\neq v)\{\textbf{if}(f(k)<v)\{search\_inc\}\ \textbf{else}\{search\_dec\}\};$$
$$found$$

The final schedule represents the program structure at the code level. In Section 4.2 we discuss how the guards of the events are removed by verifying that they follow from the explicit guards of the schedules.

4.2. Guard Propagation and Elimination Rules and Conditions

When the algorithmic structure in an Event-B model is made explicit by our scheduling language we want to be able to eliminate the guards within the events as we expect them to follow from the explicit control guards. To reason about this, it would be enough to prove that right before the execution of an event, as defined by a schedule, the corresponding guards are true. This is to say that, conditions and ordering imposed by the schedule can replace the event guards. Here we want to define a *guard elimination condition* for each event in the schedule such that that if it holds then the event guards can be eliminated safely.

Before we can define the guard elimination condition we have to make it clear how the schedule guards and ordering are related to the scheduled events. To do this, we introduce a new concept called *derived guards*. A *derived guard* is a condition that is guaranteed by the schedule to hold before the execution of an event. To illustrate derived guards, assume that we have the following simple schedule:

$$[dg]\,;\,e$$

where e represents a single event. In the above schedule dg is a derived guard for event e which is assumed to be true when control reaches the point just prior to execution of event e. A derived guard is obtained from forward propagation of the relevant schedule guards (i.e. loop or branch conditions). Forward propagation rules are discussed in Refinement Calculus [3] and here we use them for generation of derived guards. The propagation rules allow the generation of derived guards, which in turn help with the elimination of the event guards. Here we treat scheduled Event-B events as guarded actions in the form of $grd \rightarrow act$ (where grd and act represent event guards and event actions, respectively) and eliminate event guards as follows:

$$[dg]\,;(grd \rightarrow act) \sqsubseteq [dg]\,;act \quad \text{provided } dg \Rightarrow grd \tag{1}$$

The rest of this section presents the propagation rules and then presents the guard elimination condition and illustrates it through an example:

- **Rule 6 (Branch)** If dg_s is a derived guard for a conditional construct (`if..else`) with n branches then the guards can be propagated as follows:

$$[dg_s]\,;\mathbf{if}(cond_1)\{\,S_1\,\}\,\mathbf{elseif}(cond_2)\{\,S_2\,\}...\,\mathbf{else}\{\,S_{\mathrm{n}}\,\}$$
$$=$$
$$[dg_s]\,;\mathbf{if}(cond_1)\{\,[dg_1]\,;S_1\,\}\,\mathbf{elseif}(cond_2)\{\,[dg_2]\,;S_2\,\}...\,\mathbf{else}\{\,[dg_{\mathrm{n}}]\,;S_{\mathrm{n}}\,\}$$

  where

$$dg_{\mathrm{i}} \triangleq dg_{\mathrm{s}} \wedge (\neg cond_1 \wedge ... \wedge \neg cond_{\mathrm{i\text{-}1}}) \wedge cond_{\mathrm{i}}$$
$$dg_{\mathrm{n}} \triangleq dg_s \wedge \neg cond_1 \wedge ... \wedge \neg cond_{\mathrm{n\text{-}1}}$$

  and $i \in 1..n-1$. Here dg_i holds right before S_i and dg_n holds right before S_n. Depending on the structure of S_i and S_n, the derived guards may be forward propagated further.

- **Rule 7 (Loop)** A loop condition can be propagated as follows:

$$\mathbf{while}(cond)\{\,S_1\,\}$$
$$=$$
$$\mathbf{while}(cond)\{\,[cond]\,;S_1\,\}\,;[\neg cond]$$

For further propagation of derived guards we also have the following three rules:

- **Rule 8** If dg is a derived guard for a schedule and the schedule has two sub-schedules connected via a sequential composition operator:

$$[dg]\,;(S_1\,;S_2)$$

then dg is only a derived guard for the first sub-schedule:

$$[dg]\,;S_1\,;S_2$$

- **Rule 9** A schedule S preserves the condition p if:

$$[p]\,;S \sqsubseteq S\,;[p]$$

The program operators preserve conditions if their operands do:

  - $S_1\,;S_2$ preserves p provided that S_1 and S_2 preserve p.
  - **if(**$cond$**){**S_1**}else{**S_2**}** preserves p provided that $[cond]\,;S_1$ and $[\neg cond]\,;S_2$ preserve p.
  - **while(**$cond$**){**S**}** preserves p provided that $[cond]\,;S$ preserves p.

- **Rule 10** In addition to the above rule, for an atomic event e proving that the event preserves p is the same as proving invariant preservation by treating p as invariant. Invariant preservation proof obligation was discussed in Section 2.

Now that we have clear rules for obtaining derived guards we can define the guard elimination condition. Based on (1) from above, if there is a schedule with n events each event will have one guard elimination condition in the following general form:

$$I(v) \wedge dg_{\mathrm{i}} \;\Rightarrow\; G_{\mathrm{i}}$$

where *I(v)* is the model invariant and $i \in 1..n$. Also dg_i and G_i represent the derived guard and event guards of the ith event, respectively. Proving the guard elimination condition for each event allows us to replace the original event guards with derived guards safely.

By propagating schedule conditions to events as derived guards we should be able to remove the event guards. If the event guards are eliminated successfully (i.e. the execution of actions under the propagated guards preserve the model invariants) then the ordering imposed by the schedule guarantees the preservation of the model properties and establishment of the post-conditions (i.e. successful elimination of the *FINAL* event guards).

Recall the concrete schedule and model of the binary search algorithm in Section 4.1:

initialisation;
while($f(k) \neq v$)**{if**($f(k)$<v)**{**$search\_inc$**}** **else{**$search\_dec$**}}**;
$found$

By applying the forward propagation rules, the following guards (shown in brackets before each event) can be derived:

$[f(k) \neq v \wedge f(k) < v]$
Event *search_inc*
where
grd1: $f(k) < v$
then
act1: $k := (k + j + 1)/2$
act2: $i := k + 1$
End

$[f(k) \neq v \wedge \neg(f(k) < v)]$
Event *search_dec*
where
grd1: $f(k) > v$
then
act1: $k := (i + k - 1)/2$
act2: $j := k - 1$
End

$[\neg(f(k) \neq v)]$
Event $found$
where
grd1: $f(k) = v$
then
act1: $r := k$
End

Proving that the original event guards can be removed safely is trivial.

4.3. Enabledness and Guard Elimination Condition

Now that the guard elimination condition (GEC) is introduced we can show that if the enabledness condition presented in Section 3.2 for the abstract schedule refinement rules is not satisfied, we will not be able to eliminate the event guards at the final level. We illustrate this with an example. Recall the search algorithm introduced earlier. Assume that we strengthen the guards of events *search_inc* and *search_dec* by adding a new guard $i < j$. While the new events with stronger guards still satisfy the model properties, the enabledness condition cannot be satisfied:

$$f(k) \neq v \;\Rightarrow\; (f(k) < v \wedge i < j) \vee (f(k) > v \wedge i < j)$$

The explicit schedule guards presented in Section 4.1 cannot satisfy the guard elimination condition for any of the search events:

- GEC for *search_inc*: $I \wedge f(k) \neq v \wedge f(k) < v \;\Rightarrow\; f(k) < v \wedge i < j$
- GEC for *search_dec*: $I \wedge f(k) \neq v \wedge f(k) > v \;\Rightarrow\; f(k) < v \wedge i < j$

Even if we strengthen the explicit schedule conditions by adding $i < j$ to the branch condition we will not be able to satisfy the guard elimination condition for event *search_dec*. This simple example shows that if the enabledness is not preserved, we will not be able to eliminate some of the event guards.

5. Case Study: Schorr-Waite Marking Algorithm

The Schorr-Waite algorithm [13] is a well-known graph marking algorithn named after its inventors. The importance of the algorithm is due to its low memory consumption [14]. It marks all reachable nodes from a *top* node in a given graph. Instead of logging the path to the current node from the top node, the algorithm simply reverses the traversed edges and uses them for backtracking when there is no reachable unmarked node left. It was originally proposed for dealing with the problem of garbage collection when the available space is low. The algorithm has attracted substantial attention in the literature and become an interesting problem and testbed for program verification.

Apart from the importance of the algorithm in the literature, we had another reason in choosing this algorithm: the algorithm has already been modelled for binary graphs in an event-based approach and proved by Abrial in [15] without any explicit control structure. Abrial applied some *merging rules* to obtain the program structure from the model. This gives us a means to measure our approach against standard Event-B approach.

The rest of this section outlines the development of the algorithm using our approach.

5.1. The Algorithm

The Schorr-Waite algorithm is a depth first graph traversal algorithm (illustrated in Figure 5.1) and can be specified informally as follows:

- The algorithm traverses a given graph from a given *top* node
- The graph is traversed depth-first
- The algorithm marks all reachable nodes from the given *top* node
- The backtracking structure (path to the top node) is stored in the graph by reversing the traversed edges
- Each time the algorithm backtracks it reverses the edge to its original direction

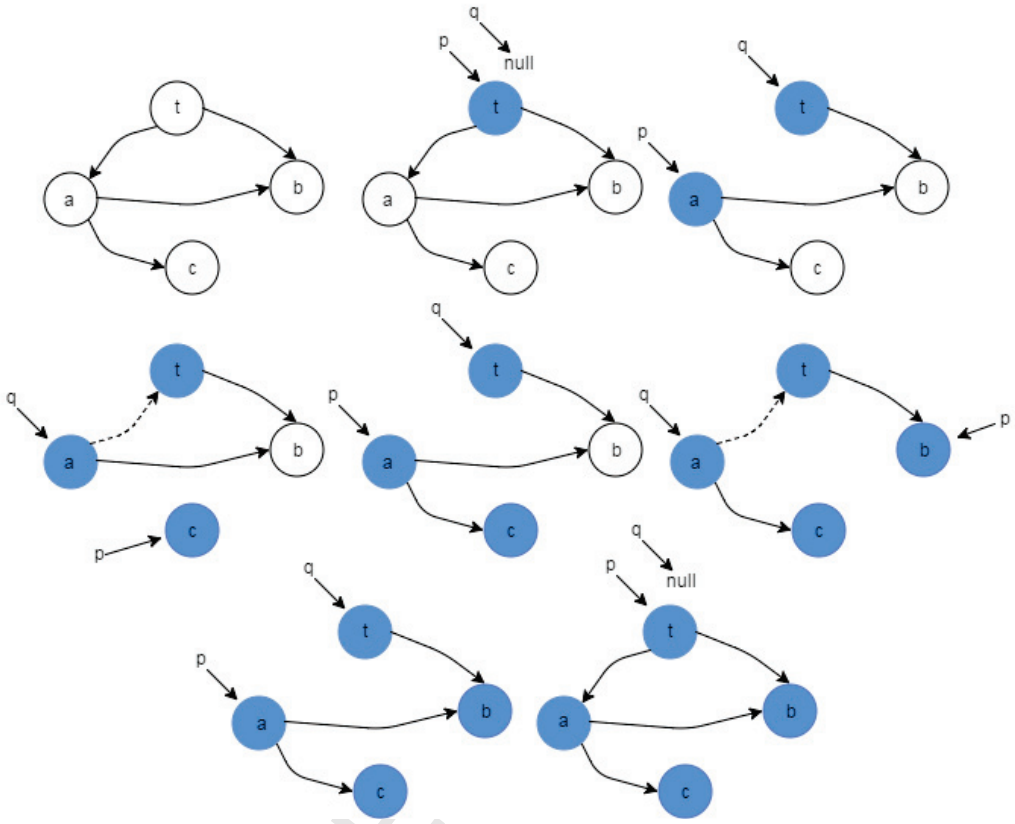


Figure 3: An illustration of how the Schorr-Waite algorithm marks a graph from node t. p and q are two auxiliary pointers to current and previous nodes, respectively. Dashed arrows represent the reversed edges used in the backtracking phase. The most recent traversed edge is represented using p and q and no explicit arrow is shown. The rest of the traversed edges are represented within the graph.

The algorithm has four main functional properties that a correct implementation should guarantee for any given unmarked graph (with a set of finite nodes) and any arbitrary *top* node:

1. The algorithm will mark all reachable nodes from the *top* node.
2. The algorithm will only mark reachable nodes from the *top* node.
3. The algorithm will terminate.
4. Upon termination, the graph structure is the same as its original structure.

5.2. The Development of the Algorithm

As mentioned before, the algorithm has previously been modelled by Abrial [15] for a binary graph to show how a sequential program can be constructed using an event-based approach, however the model was not refined to a concrete program.

We borrowed the abstract specification of the algorithm from Abrial's work and developed and refined our version (where each node may have an arbitrary number of children) to a concrete level based on it. Here we omit most of model details (invariants and event details) and explain it briefly and concentrate on the refinement of the algorithmic structure using our scheduling language and rules [1].

5.2.1. Abstract Specification

At the abstract level the graph is specified using a constant binary relation (g) on a set N of nodes. To find all reachable nodes in the graph from the top node t it would be enough to calculate the image of $\{t\}$ under the transitive closure of g. Therefore the abstract model needs to have only one event (*marked*) whose execution will mark all the reachable nodes in one shot. The associated schedule with this specification is simply:

$$initialisation\,;marked$$

The context of the abstract model is as follows:

SETS N
CONSTANTS g, c, t
AXIOMS
axm1: $g \in N \leftrightarrow N$
axm2: $c \in N \leftrightarrow N$
axm3: $t \in N$
axm4: $\forall s.(s \subseteq N \Rightarrow s \subseteq c[s])$
axm5: $\forall s, x, y.(s \subseteq N \wedge x \mapsto y \in g \wedge x \in c[s] \Rightarrow y \in c[s])$
axm6: $\forall s.(s \subseteq N \wedge g[s] \subseteq s \Rightarrow c[s] \subseteq s)$
axm7: $finite(N)$

[1] The full Event-B model of the algorithm can be found at: `http://users.ecs.soton.ac.uk/md5g11/SchorrWaite/`

In the context, g represents the graph and c is the transitive closure of g. Axioms $axm4$, $axm5$, and $axm6$ specify the properties of the transitive closure (c) needed for specifying the desirable outcome of the algorithm. The *marked* event which calculates all the reachable nodes from top node t using the image of the transitive closure under t in one step has the following definition:

Event *marked*
then
act1: $r \coloneqq c[\{t\}]$
End

5.2.2. The First Refinement: Introducing the algorithm

From this refinement we gradually introduce the algorithm and refine it to a concrete level. Here a new event (*marking*) is introduced to compute the reachable nodes and mark them (add them to set b) gradually by its repetition. The *marking* event selects the next unmarked reachable node from the set of marked nodes (b) non-deterministically. By applying rule 2 from 3.2 we can refine the abstract schedule:

$$initialisation\,;marking^*\,;marked$$

5.2.3. The Second Refinement: Introducing the backtracking

This refinement specifies an important feature of the algorithm: backtracking. The backtracking structure is specified using an injective function f. Function f allows the algorithm to backtrack to the previous node when it cannot traverse further. Also a pointer to the current node p and a flag n are introduced. If n is true then all reachable nodes are marked. Two new events *backtracking* and *termination* are added to the model. The *backtracking* event changes the current node to the previous node if all the children of p are marked and $p \neq t$ and *termination* changes the flag n to $TRUE$ if $p = t$ and all of the t children are marked or it does not have any. The *marking* event is also refined to update the backtracking structure when it traverses from the current node to its child. By applying Rule 3 (loop body extension) we get the following schedule:

$$initialisation\,;(termination \,\square\, backtracking \,\square\, marking)^*\,;marked$$

5.2.4. The Third and Fourth Refinements: Data Refinement

The third refinement adds a boolean attribute *marked* to each node. This is specified as a total function from N to $BOOL$. The algorithm will set marked to true if the node is reached.

The fourth refinement changes the representation of the graph to a total function from nodes to a sequence of nodes. The sequence represents the children of each node. In the third and fourth refinements we only perform data refinement. If in a level only data refinement is performed then the schedule remains the same as the abstract one:

$$initialisation\,;(termination \,\square\, backtracking \,\square\, marking)^*\,;marked$$

5.2.5. The Fifth Refinement: Deterministic Choice of the Next Node

Until now the selection of the next child was specified non-deterministically. Now that we have the children of the current node in an ordered list, the selection of the next node can be specified deterministically. To facilitate this, we add a new attribute (chV) to N to count the number of visited children of each node. This is specified by a total function from N to $\mathbb{Z}$. The *marking* event is refined to choose the chV^{th} child of p as the next node if it is not already marked and increases the chV of p by 1. A new event *nextChild* is also introduced to increase the chV of p by 1 in case that the chV^{th} child is already marked. The algorithm now backtracks if chV of p is equal to the number of children of node p if $p \neq t$. If chV of p is equal to the number of children of node p and also $p = t$ then the flag n is set to $TRUE$. By applying Rule 3 (loop body extension), the schedule from the last level can be refined:

$$\begin{gathered} initialisation; \\ (nextChild \,\square\, termination \,\square\, backtracking \,\square\, marking)^*; \\ marked \end{gathered}$$

5.2.6. The Sixth Refinement: Data Refinement

This is another data refinement. In this level we introduce a constant node *null*. *null* is not in the graph. We also introduce a new pointer q which points to the parent of node p if $p \neq t$. If $p{=}t$ then $q{=}null$. The *marked* and *backtracking* events are extended to update q where appropriate. The corresponding schedule to this level is the same as previous one:

$$\begin{gathered} initialisation; \\ (nextChild \,\square\, termination \,\square\, backtracking \,\square\, marking)^*; \\ marked \end{gathered}$$

5.2.7. Seventh Refinement: Introduction of Concrete Program Structure

In this refinement finally we store the backtracking structure within the graph itself and remove it from the algorithm. We add a new attribute to N called cg for storing the list of a node's children. Events *marking* and *backtracking* are refined to reverse the traversed or backtracked edges. Now that we have the concrete data structure and the actions of events are deterministic, non-deterministic choices or loops in the schedule can be refined to a concrete branch or loop. By applying Rules 4 and 5 given in 3.2 we will have the final concrete schedule shown in Figure 4. The events of the concrete model of the algorithm are as follows:

Event *nextChild*
where
grd1: $cg(p) \neq \varnothing$
grd2: $chV(p) < card(cg(p))$
grd3: $n = FALSE$
grd4: $marked(cg(p)(chV(p))) = TRUE$
then
act1: $chV(p) := chV(p) + 1$
End

Event *termination*
where
grd1: $n = FALSE$
grd2: $(cg(p) \neq \varnothing \wedge chV(p) = card(cg(p))) \vee cg(p) = \varnothing$
grd3: $p = t$
then
act1: $n := TRUE$
End

Event *marking*
any z
where
grd1: $cg(p) \neq \varnothing$
grd2: $chV(p) < card(cg(p))$
grd3: $n = FALSE$
grd4: $z = cg(p)(chV(p))$
grd5: $marked(z) \neq TRUE$
then
act1: $marked(z) := TRUE$
act2: $p := z$
act3: $chV(p) := chV(p) + 1$
act4: $path := path \cup \{z\}$
act5: $q := p$
act6: $cg(p)(chV(p)) := q$
End

Event *backtracking*
where
grd1: $n = FALSE$
grd2: $(cg(p) \neq \varnothing \wedge chV(p) = card(cg(p))) \vee cg(p) = \varnothing$
grd3: $p \neq t$
then
act1: $p := q$
act2: $path := path \setminus \{p\}$
act3: $q := cg(q)(chV(q)\text{=}1)$
act4: $cg(q)(chV(q) - 1) := p$
End

```
initialisation;
while(n = FALSE){
    if(chV(p) = card(cg(p)))
    {
        if(p=t){ termination } else{ backtracking }
    }
    else
    {
        if(marked(p)=TRUE){ nextChild } else{ marking }
    }
};
marked
```

Figure 4: The final schedule representing the concrete algorithmic structure of the Schorr-Waite Algorithm

To ensure the correctness of the program structure given in Figure 4 we have to apply forward propagation rules and eliminate the original event guards. The following five conditions (one for each guarded event in the schedule) guarantees the safe elimination of the guards:

1. GEC for **termination** event:

   $I \wedge n = FALSE \wedge chV(p) = card(cg(p)) \wedge p = t \;\Rightarrow\; grd1 \wedge grd2 \wedge grd3$

2. GEC for **backtracking** event:

   $I \wedge n = FALSE \wedge chV(p) = card(cg(p)) \wedge \neg(p = t) \;\Rightarrow\; grd1 \wedge grd2 \wedge grd3$

3. GEC for **nextChild** event:

   $I \wedge n = FALSE \wedge \neg(chV(p) = card(cg(p))) \wedge marked(p) = TRUE \;\Rightarrow\; grd1 \wedge grd2 \wedge grd3 \wedge grd4$

4. GEC for **marking** event:

   $I \wedge n = FALSE \wedge \neg(chV(p) = card(cg(p))) \wedge \neg(marked(p) = TRUE) \;\Rightarrow\; grd1 \wedge grd2 \wedge grd3 \wedge grd4$

5. GEC for **marked** event:

   $I \wedge \neg(n = FALSE) \;\Rightarrow\; grd1$

In the above conditions I represents model invariants and the right hand side of each implication ($grd1 \wedge grd2 \wedge ...$) is referring to conjunction of guards of the respective event. All the above conditions can be trivially proved.

The Event-B model and proof of the Schorr-Waite algorithm presented in this section is similar to Abrial's model and proof presented in [15]. The main difference between the two developments of the algorithm is the program derivation method that is employed. In Abrial's work two *merging rules* are used in order to derive the program structure (conditionals and loops) from the concrete model. In contrast, we use our scheduling language in order to derive the program structure in successive steps. Our abstract scheduling language (non-deterministic choice and iteration) allows us to capture the control flow between events easier. It also enables us to refine the abstract schedule and establish a relation between the schedules in different levels of abstraction. An immediate benefit of using scheduled Event-B is gaining a better insight into the relationship between different atomic Event-B events from the abstract level.

When using Abrial's approach for developing sequential programs with Event-B, adopting different refinement strategies can lead to different program structures for the same problem. This means that the refinement strategy and the program structure are coupled [16]. Following our approach, the modeller should think about the control flow from the very abstract level and make it explicit. Refining the schedule using the provided refinement rules will assist the modeller in deriving the concrete program in refinement steps and make it easier to identify the correct program structure.

6. Related and Future Work

Abrial in [15] proposed a method for developing sequential programs in an event-based approach. The approach is based on two *merging rules* for defining conditional statements and loops. Merging rules may be effective in many cases for derivation of sequential programs from Event-B models, however, they do not help to derive the control flow in refinement steps.

Making the control flow for Event-B models explicit has been explored by others as well. Hallerstede in [17] introduced a method for specifying structured models in Event-B. The notation that is used by the method drives the generation of the necessary proof obligations and also specifies the structure of a model. The method has a corresponding graphical notation

in order to help the modeller to understand the structure of the model more easily. The notation allows the modeller to make the control flow explicit however it does not support concrete branches and loops. Illiasov in [18] extended Event-B with expressions called *flows* specifying the ordering between events. A flow allows the modeller to specify sequential composition, loop and choice. The approach uses the refinement of traces as a definition of the refinement relation. The aforementioned approaches do not provide rules for introducing program structures in stepwise refinement. Boström in [19] proposed a method for introducing control flow constructs to Event-B. The method provides a set transformer based semantics for Event-B and a scheduling language for describing the control flow in a model. The scheduling language has control constructs like branch and loop however it lacks explicit conditions for those constructs and does not introduce any rules for stepwise refinement of the schedule. Schneider et al. in [20] proposed a combined method by using CSP to provide explicit ordering for an Event-B model. Fürst et al. in [21] presented an approach for generating sequential code from Event-B model. In order to facilitate code generation a scheduling language for specifying event ordering is provided. The approach uses explicit program counters and auxiliary events for proving the correctness of the scheduled model. The auxiliary events and program counters can be problematic since for a model with complex control structure we may end up with a large number of control events and invariants which are necessary for proof. Edmunds and Butler in [22] proposed an extension to Event-B in order to facilitate code generation. The control flow in the model should be made explicit prior to code generation using a scheduling language. The approach does not support incremental introduction of control flow. Fathabadi et al. in [23] introduced event refinement structure (ERS). ERS is a graphical notation for explicit representation of the relationship between abstract events and new events in a refinement level. It also provides a way for imposing explicit sequencing between events. ERS is designed for distributed systems and does not target sequential programs.

Research for generating code from formal models is not limited to Event-B. There is much research in the literature focusing on the derivation of executable code from formal models in languages like VDM [24] and Z [25].

VDM is supported by a tool called VDMTools which supports automatic C++ and Java code generation [26]. Overture [27] is an open source platform for supporting the construction and analysis of VDM formal models. Jørgensen et al. in [28] introduced a code generation platform for VDM in

the Overture tool. The platform transforms a model to an Intermediate Representation (IR) in order to make the task of code generation easier. Since the IR is independent from any programming language, it can be later used for generation of code in the different languages. Rafsanjani and Colwill in [29] and Fukagawa et al. in [30] presented an approach for structural mapping from Object-Z [31] (an object-oriented extension to Z) to C++ code. The main purpose of the approach was to aid the developer in obtaining code. In [32] an approach for animating Object-Z specifications using C++ is presented. This approach extends the aforementioned structural mapping by covering a larger subset of the Object-Z language. In [33] an approach for mapping a subset of Object-Z specifications to skeletal Java classes is introduced. The approach tries to take advantage of code contracts in order to ensure the correctness of the generated code. However the contracts are encoded in the program code as condition of branches and no static analysis is performed. *Circus* [34, 35] is a unified language based on Z, CSP and the refinement calculus for specifying, designing and programming concurrent programs introduced by Woodcock and Cavalcanti. It combines Z and CSP in a way that is suitable for refinement. A number of tools have been developed to provide support for refinement and code generation for Circus.

In the future we would like to extend our previous work [36, 37] where we introduced a method for transforming Event-B models to Dafny [38] code contracts by augmenting it with the algorithmic details provided by the scheduling language introduced in this paper. Our previous method transforms abstract Event-B models to simple Dafny code contracts (i.e. method's pre- and post-conditions) in a way that any implementation that can be verified against the generated contracts is considered as a correct implementation of the abstract model. Using this approach the user should come up with a proper program structure at code level and try to prove things like termination there. This approach works fine if there is an implementation that can be verified against the generated pre- and post-conditions with no or minimal efforts in terms of providing more assertions (e.g. loop invariants). However, if the implementation involves complex program structures such as nested loops or branches, then this approach is not effective since a large part of the development and verification is left for the code level. We want to employ our scheduling language to perform algorithmic refinement in an abstract level and then transform a model to Dafny code and contracts with algorithmic structure. At the Dafny level we aim to avoid re-verifying the properties that are already proved in Event-B and our scheduling language

and only focus to verify that the generated code correctly implements the lowest scheduled Event-B model. This will include correctly decomposing the parallel actions of each event into sequential compositions of assignments. We also like to extend our work by providing rules for automatic generation of schedule conditions in a way that event guards can be eliminated.

7. Conclusion

In this paper we introduced an approach that allows us to mix program refinement and data refinement using the existing Event-B tool support for representing and proving the data refinement. We augmented Event-B with a *scheduling language* language to make the control flow and algorithmic structure explicit.

The scheduling language and the schedule refinement rules allow the modeller to derive the program structure in a stepwise manner. The scheduling language allows the modeller to introduce loops and non-deterministic choices at the abstract level. It also has familiar control constructs like if-statements and while-loops that will replace the non-deterministic structures at the concrete level. We presented a number of rules which will assist the modeller in refining an abstract schedule towards a concrete one. We also explained how guard elimination condition can guarantee that the events in the schedule will follow the explicit control flow and how we can remove original event guards.

We validated our approach by developing and verifying the well-known Schorr-Waite graph marking algorithm using Event-B and our scheduling language. The explicit control flow defined by the scheduling language allowed us to gain a better insight on the ordering of the events and the algorithmic structure from the abstract level and the given rules in 3.2 and 4 helped us in derivation of the final program structure in a stepwise manner.

Acknowledgement

We would like to thank anonymous reviewers for their careful reading of the paper and their comments and suggestions.

References

[1] J.-R. Abrial, Modeling in Event-B: system and software engineering, Cambridge University Press, 2010.

[2] J.-R. Abrial, M. Butler, S. Hallerstede, T. Hoang, F. Mehta, L. Voisin, Rodin: an open toolset for modelling and reasoning in Event-B, International Journal on Software Tools for Technology Transfer 12 (6) (2010) 447–466.

[3] R.-J. Back, J. Wright, Refinement calculus: a systematic introduction, springer Heidelberg, 1998.

[4] C. Morgan, The specification statement, ACM Transactions on Programming Languages and Systems (TOPLAS) 10 (3) (1988) 403–419.

[5] J. M. Morris, A theoretical basis for stepwise refinement and the programming calculus, Science of Computer programming 9 (3) (1987) 287–306.

[6] E. W. Dijkstra, Guarded commands, nondeterminacy and formal derivation of programs, Communications of the ACM 18 (8) (1975) 453–457.

[7] R. J. R. Back, F. Kurki-Suonio, Distributed cooperation with Action Systems, ACM Trans. Program. Lang. Syst. 10 (4) (1988) 513–554. doi:10.1145/48022.48023.
URL http://doi.acm.org/10.1145/48022.48023

[8] M. Butler, Incremental design of distributed systems with Event-B, Engineering Methods and Tools for Software Safety and Security 22 (2009) 131.

[9] J. von Wright, The lattice of data refinement, Acta Informatica 31 (2) (1994) 105–135. doi:10.1007/BF01192157.
URL http://dx.doi.org/10.1007/BF01192157

[10] J. M. Morris, Laws of data refinement, Acta Informatica 26 (4) (1989) 287–308.

[11] J. von Wright, From Kleene Algebra to Refinement Algebra, Springer Berlin Heidelberg, Berlin, Heidelberg, 2002, pp. 233–262.

[12] N. Wirth, Extended Backus-Naur Form (ebnf), ISO/IEC 14977 (1996) 2996.

[13] H. Schorr, W. M. Waite, An efficient machine-independent procedure for garbage collection in various list structures, Communications of the ACM 10 (8) (1967) 501–506.

[14] R. Bubel, The Schorr-Waite algorithm, in: Verification of Object-Oriented Software. The KeY Approach, Springer, 2007, pp. 569–587.

[15] J.-R. Abrial, Event based sequential program development: Application to constructing a pointer program, in: FME 2003: Formal Methods, Springer, 2003, pp. 51–74.

[16] S. Hallerstede, M. Leuschel, Experiments in program verification using Event-B, Form. Asp. Comput. 24 (1) (2012) 97–125.

[17] S. Hallerstede, Structured Event-B models and proofs, in: Abstract State Machines, Alloy, B and Z, Springer, 2010, pp. 273–286.

[18] A. Iliasov, On Event-B and control flow, in: TECHNICAL REPORT SERIES, School of Computing Science Newcastle University, 2010, p. 19.

[19] P. Boström, Creating sequential programs from Event-B models., in: IFM, Springer, 2010, pp. 74–88.

[20] S. Schneider, H. Treharne, H. Wehrheim, A CSP approach to control in Event-B, in: Integrated formal methods, Springer, 2010, pp. 260–274.

[21] A. Fürst, T. S. Hoang, D. Basin, K. Desai, N. Sato, K. Miyazaki, Code generation for Event-B, in: Integrated Formal Methods, Springer, 2014, pp. 323–338.

[22] A. Edmunds, M. Butler, Tasking Event-B: An extension to Event-B for generating concurrent code, Programming Language Approaches to Concurrency and Communication-cEntric Software (2011) 1.

[23] A. S. Fathabadi, M. Butler, A. Rezazadeh, Language and tool support for event refinement structures in Event-B, Formal Aspects of Computing (2014) 1–25.

[24] V. S. Alagar, K. Periyasamy, Vienna Development Method, in: Specification of Software Systems, Texts in Computer Science, Springer London, 2011, pp. 405–459.

[25] J. M. Spivey, J. Abrial, The Z notation, Prentice Hall Hemel Hempstead, 1992.

[26] J. Fitzgerald, P. G. Larsen, S. Sahara, et al., VDMTools: advances in support for formal modeling in VDM, ACM Sigplan Notices 43 (2) (2008) 3.

[27] P. G. Larsen, N. Battle, M. Ferreira, J. Fitzgerald, K. Lausdahl, M. Verhoef, The Overture initiative integrating tools for VDM, SIGSOFT Softw. Eng. Notes 35 (1) (2010) 1–6. doi:10.1145/1668862.1668864.
URL http://doi.acm.org/10.1145/1668862.1668864

[28] P. W. V. Jørgensen, M. Larsen, L. D. M. D. Couto, A code generation platform for VDM, University of Newcastle-upon-tyne. Computing Science. Technical Report Series.

[29] G.-H. B. Rafsanjani, S. J. Colwill, From Object-Z to C++: A Structural Mapping, Springer London, London, 1993, pp. 166–179.

[30] M. Fukagawa, T. Hikita, H. Yamazaki, A mapping system from Object-Z to C++, in: Proceedings of 1st Asia-Pacific Software Engineering Conference, 1994, pp. 220–228. doi:10.1109/APSEC.1994.465258.

[31] R. Duke, G. Rose, G. Smith, Object-Z: A specification language advocated for the description of standards, Computer Standards and Interfaces 17 (5–6) (1995) 511–533.

[32] M. Najafi, H. Haghighi, An approach to animate Object-Z specifications using C++, Scientia Iranica 19 (6) (2012) 1699 – 1721. doi:http://dx.doi.org/10.1016/j.scient.2012.06.021.
URL http://www.sciencedirect.com/science/article/pii/S1026309812001423

[33] S. Ramkarthik, C. Zhang, Generating Java skeletal code with design contracts from specifications in a subset of Object-Z, in: 5th IEEE/ACIS International Conference on Computer and Information Science and 1st IEEE/ACIS International Workshop on Component-Based Software Engineering,Software Architecture and Reuse (ICIS-COMSAR'06), 2006, pp. 405–411.

[34] J. Woodcock, A. Cavalcanti, The Semantics of Circus, Springer Berlin Heidelberg, Berlin, Heidelberg, 2002, pp. 184–203.

[35] J. Woodcock, A. Cavalcanti, A concurrent language for refinement., in: IWFM, Vol. 1, 2001, p. 5th.

[36] M. Dalvandi, M. J. Butler, A. Rezazadeh, Transforming Event-B models to Dafny contracts, ECEASST 72.
URL http://journal.ub.tu-berlin.de/eceasst/article/view/1021

[37] M. Dalvandi, M. Butler, A. Rezazadeh, From Event-B models to Dafny code contracts, in: Fundamentals of Software Engineering, Springer, 2015, pp. 308–315.

[38] K. R. M. Leino, Dafny: An automatic program verifier for functional correctness, in: Logic for Programming, Artificial Intelligence, and Reasoning, Springer, 2010, pp. 348–370.