Pontus Boström

Creating sequential programs from Event-B models

Turku Centre *for* Computer Science

TUCS Technical Report
No 955, August 2009

Creating sequential programs from Event-B models

Pontus Boström
Department of Information Technology,
Åbo Akademi University,
Joukainengatan 3-5,
FIN-20520 Åbo
`pontus.bostrom@abo.fi`

TUCS Technical Report
No 955, August 2009

Abstract

Event-B is an emerging formal method with good tool support for various kinds of system modelling. However, efficient implementation of event based specifications created in Event-B as imperative programs has not been thoroughly explored. This paper explores methods to create efficient sequential code from Event-B models. The methods are based on a scheduling language for describing the flow of control in programs. The aim is to be able to express schedules of events, reasoning about their correctness and for creating and verifying implementation patterns. The conclusion is that using patterns, it is feasible to derive efficient sequential code from event based specifications in many cases. However, the extra proof obligations required in the process might sometimes make this approach infeasible.

Keywords: Event-B, Implementation, Sequential program development, Scheduling, Patterns

TUCS Laboratory
Distributed Systems Design Laboratory

1 Introduction

Event-B [3] has emerged as a well known method for high-level specification of a wide variety of different types of systems. It has good tool support in the form of theorem provers [3], animators [13] and model checkers [13] to analyse specifications. The specifications can also be developed and analysed in a stepwise manner using the notion of refinement. However, when developing software, the final step from specification to executable program code has not been thoroughly explored.

An Event-B model consists of a set of events inside a **do od** -loop, where one enabled event is non-deterministically chosen for execution in each iteration of the loop. If there are no enabled events, the loop terminates. To implement an Event-B model, this loop is translated in some manner to an (imperative) programming language. In [1, 2] Abrial gives a method for development of sequential programs in Event-B and patterns for introducing flow control, such as while-loops and if-statements, to obtain the final imperative program. Those patterns work well in many cases, but they fairly limited in the type of sequential behaviour that can be introduced. A translator has also been developed for translating Event-B to C [15]. This translator translates events to their corresponding C constructs and directly creates a loop body where one of the events is executed in each iteration of the loop. This approach always works, but the programs might be very inefficient. By inefficient we here mean that the program runs slower than necessary, e.g., due to unnecessary tests and branches because of the simplistic control flow. Event-B is based on action systems [4, 5]. Implementation strategies for action systems and Event-B models have also been investigated for parallel programs e.g. in [4, 11].

This paper investigates methods to create efficient sequential programs from event based specifications. By creation of a sequential program, we mean that flow control constructs such as while-loops, if-statements and sequential composition have been introduced to schedule the events from the Event-B model. The methods are based on using a scheduling language to describe the flow of control. If the events are executed according to the schedule and the schedule has been verified, the resulting sequential program is a refinement of the original event loop. The main goal is to show how to verify scheduling of events and to be able to actually derive the conditions under which the schedule is correct. We also investigate how to develop and reason about reusable patterns to simplify development of sequential programs in practice. To illustrate the approach, the methods are, for example, used to prove the loop introduction pattern in [1, 2] correct. A number of additional patterns are also presented.

The paper starts with an overview of Event-B. Event-B has no fixed semantics and we first present a semantics based on a set transformer semantics of events similar to [12] and action systems. This semantics is compatible with the proof obligations generated by the Event-B tools. We then present sequential program construction using Event-B as described in [1, 2]. To create sequential programs from the Event-B models, a scheduling language is then developed. We then show how scheduling events according to the schedule will lead to a refinement of the original model. Patterns for verified schedules, as well as creation of executable program statements from the scheduled events are also presented. A small example applying the method is then finally given to illustrate our approach.

2 Event-B

An Event-B model is referred to as a *machine* [3]. A machine consists of a set of *variables* constituting the state of the model, an *invariant* that describes the valid states and a collection of *events* that describes the evolution of the state of the model. The model can also use a *context machine*, with *constant* and *set* definitions. The properties of these constants and sets are given by a set of *axioms.*

2.1 Events as set transformers

The events in Event-B can be viewed as set or predicate transformers [6, 12]. The presentation here follows the presentation of set transformer semantics of events in [12]. The state space of an Event-B model is modelled as the Cartesian product of the types of the variables. Variables $v_1, \ldots, v_n$ having types $\Sigma_1, \ldots, \Sigma_n$ give the state space $\Sigma = \Sigma_1 \times \ldots \times \Sigma_n$.

An event E has the form $E \mathrel{\hat{=}} \textbf{when } G(v,c) \textbf{ then } v :| S(v', v, c) \textbf{ end}$. Here v denotes the variables, c the constants, $G(v,c)$ denotes the guard of the event and $v :| S(v', v, c)$ denotes a non-deterministic assignment to the variables. Whenever $G(v,c)$ is true the substitution $v :| S(v', v, c)$ can be executed. The substitution assigns variables v any value v' such that S holds. The events of the machine are then considered to be inside a loop, where one event is executed in each iteration of the loop as long as there are enabled events. To describe the valid subset of the state space an invariant I is used. A model has a special initialisation event *initialisation*, which has a a substitution of the form $v :| A$. We have the following definitions:

$$\begin{array}{rcl} \Sigma & = & \{v | \top\} \\ i & = & \{v | I(v,c)\} \\ g & = & \{v | G(v,c)\} \\ s & = & \{v \mapsto v' | S(v, v', c)\} \\ a & = & \{v' | A(v', c)\} \end{array} \tag{1}$$

The sets i and g describe the sets where the invariant and the guard G holds, respectively. The relation s describes the possible before-after states that can be achieved by the assignment. Note that the initialisation results in a set a instead of a relation, since it does not depend on the previous values of the variables. In this paper, we do not consider properties of constants c separately, as it is not important on this level of reasoning. The axioms that describe the properties of the constants are here considered to be part of the invariant.

To simplify the definitions and increase readability we will introduce the following notation. The symbol true will be used to denote the complete state space of an Event-B model. In the above definition we have that $\mathsf{true}_\Sigma = \Sigma$. We will use negation to denote the complement of a set $\neg q \mathrel{\hat{=}} \mathsf{true} \backslash q$ [1]. For completeness false is defined as $\mathsf{false} \mathrel{\hat{=}} \varnothing$. This is similar to the approach in [6].

We give the semantics of the events as set transformers. A set transformer applied to a set q describes the set of states from where the set transformer will reach a state in q (cf. weakest precondition semantics of programs). We have

[1] $\backslash$ denotes set subtraction

the following set transformers

$$\begin{aligned}
&[g](q) \mathrel{\hat{=}} \neg g \cup q && \text{(Assumption)} \quad (2)\\
&\{g\}(q) \mathrel{\hat{=}} g \cap q && \text{(Assertion)} \quad (3)\\
&[s](q) \mathrel{\hat{=}} \{v | s[\{v\}] \subseteq q\} && \text{(Non} - \text{deterministic update)} \quad (4)\\
&(S_1 \sqcap S_2)(q) \mathrel{\hat{=}} S_1(q) \cap S_2(q) && \text{(Non} - \text{deterministic choice)} \quad (5)\\
&S_1; S_2(q) \mathrel{\hat{=}} S_1(S_2(q)) && \text{(Sequential composition)} \quad (6)\\
&S^{\omega}(q) \mathrel{\hat{=}} \mu X.(S; X \sqcap \mathsf{skip})(q) && \text{(Strong iteration)} \quad (7)\\
&S^{*}(q) \mathrel{\hat{=}} \nu X.(S; X \sqcap \mathsf{skip})(q) && \text{(Weak iteration)} \quad (8)\\
&\mathsf{skip}(q) \mathrel{\hat{=}} q && \text{(Skip)} \quad (9)\\
&\mathsf{magic}(q) \mathrel{\hat{=}} [\mathsf{false}](q) && \text{(Magic)} \quad (10)\\
&\mathsf{abort}(q) \mathrel{\hat{=}} \{\mathsf{false}\}(q) && \text{(Abort)} \quad (11)
\end{aligned}$$

The sequential composition ; has higher precedence than $\sqcap$ here. The set transformers here are similar to the predicate transformers in e.g. [6]. The first four set transformers (2)-(6) also occur in [12]. The two set transformers (7) and (8) model iteration of a set transformer S using the least and greatest fixpoint [6, 7], respectively. The three last set transformers are important special cases: skip is a set transformer that does nothing, $\mathsf{magic}(q)$ is always true and $\mathsf{abort}(q)$ is always false.

We have the following rules that will be useful later. Here g and h denotes two sets. These rules are easy to prove directly from the definitions and proofs can also be found in [6].

$$\begin{array}{rclrcl}
\{g\}; \{h\} & = & \{g \cap h\} & [g]; [h] & = & [g \cap h] \\
\{g\} \sqcap \{h\} & = & \{g \cap h\} & [g] \sqcap [h] & = & [g \cup h] \\
\mathsf{abort} \sqcap S & = & \mathsf{abort} & \mathsf{magic} \sqcap S & = & S \\
\mathsf{abort}; S & = & \mathsf{abort} & \mathsf{magic}; S & = & \mathsf{magic}
\end{array} \qquad (12)$$

The guard g of a set transformer denotes the states where it will not behave miraculously (establish false) and the termination guard t denotes the states where it will terminate properly (reach a state in the state space).

$$\mathsf{g}(S) = \neg S(\mathsf{false}) \qquad (13)$$

$$\mathsf{t}(S) = S(\mathsf{true}) \qquad (14)$$

These functions have the following useful properties:

$$\begin{array}{rclrcl}
\mathsf{g}(\{g\}) & = & \mathsf{true} & \mathsf{g}([g]) & = & g \\
\mathsf{t}(\{g\}) & = & g & \mathsf{t}([g]) & = & \mathsf{true} \\
\mathsf{t}(S_1 \sqcap S_2) & = & \mathsf{t}(S_1) \cap \mathsf{t}(S_2) & \mathsf{g}(S_1 \sqcap S_2) & = & \mathsf{g}(S_1) \cup \mathsf{g}(S_2)
\end{array} \qquad (15)$$

The proofs are again easy to do directly from the definitions.

Using the definitions in (1) and the set transformers in (2) and (4), an event $E \mathrel{\hat{=}}$ **when** $G(v, c)$ **then** $v :| S(v', v, c)$ **end** can be modelled as the set transformer $[E] \mathrel{\hat{=}} [g]; [s]$ [12]. Note that the proof obligations for Event-B guarantee that $\mathsf{g}([E]) = g$ and $\mathsf{t}([E]) = \mathsf{true}$ for all events [12]. The initialisation event *initialisation* is modelled by the set transformer $[\mathit{initialisation}] \mathrel{\hat{=}} [a]$. For simplicity, we usually use the event E to directly denote the set transformer $[E]$ when it is clear from the context which one is meant.

A traditional Event-B model can be modelled as an action system. Action systems [4, 5] were originally developed to reason about distributed and reactive systems, but has since been used for a wide variety of systems. An action system essentially consists of a list of variables, an initialisation of the variables and a set of actions given as set transformers. The variables form the state space of the system and the actions describe the evolution of the system. The actions are inside a **do od** -loop and one action is non-deterministically chosen for execution in each iteration of the loop. If there are no enabled actions the loop terminates. Consider an Event-B model $\mathcal{M}$ that has variables v, initialisation event *initialisation* and events $E_1, \ldots, E_n$. The corresponding action system $\mathcal{M}_A$ is then:

$$\mathcal{M}_A \mathrel{\hat{=}} |[\ \mathbf{var}\ v;\ \mathbf{init}\ [\mathit{initialisation}];\ \mathbf{do}\ [E_1] \sqcap \ldots \sqcap [E_n]\ \mathbf{od}\ ]| \tag{16}$$

The action system $\mathcal{M}_A$ has a list of variables v corresponding to the variables in the Event-B model. The initialisation event, *initialisation*, of the Event-B model becomes the initialisation of the action system. The **do od** -loop denotes non-deterministic execution of the events $[E_1] \sqcap \ldots \sqcap [E_n]$. In each execution of the loop body, one event is non-deterministically chosen. In terms of the set transformers presented earlier, the semantics of the loop is defined as [7]:

$$\mathbf{do}\ S\ \mathbf{od}\ \mathrel{\hat{=}} S^{\omega}; [\neg \mathsf{g}(S)] \tag{17}$$

This means S is executed zero or more times. The assumption at the end ensures that the loop does not terminate before the guard of S is false. Action systems have been explored thoroughly before. We base this work mainly on the algebraic rules for loops (and thus action systems) in [7].

2.2 Invariants

The invariant describes the valid part of the state space. From the Event-B proof obligations we have that the initialisation *initialisation* and every event in E preserves an invariant i, i.e. we will not reach a state where the invariant does not hold.

$$a \subseteq i \tag{18}$$

$$i \subseteq E(i) \tag{19}$$

This gives the following properties on the set transformer level [6]:

$$\mathit{initialisation} = \mathit{initialisation}; \{i\} \tag{20}$$

$$\{i\}; E = \{i\}; E; \{i\} \tag{21}$$

These assumptions can be used to introduce invariant assertions into loops as shown in Lemma 1. The assertions will be useful later for refinement and for creating sequential statements from the loops.

Lemma 1. *Using the assumption (21) above then* $\{i\}; E^{\omega} = (\{i\}; E; \{i\})^{\omega}$.

Proof. We have the induction rule (Corollary 8 in [7]): $S; X \sqcap T \sqsubseteq X \Rightarrow S^{\omega}; T \sqsubseteq$

X. Assume $X \mathrel{\hat{=}} (\{i\}; E; \{i\})^\omega$, $S \mathrel{\hat{=}} E$ and $T \mathrel{\hat{=}} \{i\}$

$$\begin{array}{ll}
 & \{i\}; E^\omega \\
\sqsubseteq & \{\text{Assumption (21)} : \{i\}; E = \{i\}; E; \{i\}\} \\
 & E^\omega; \{i\} \\
\sqsubseteq & \{\text{Induction}\} \\
 & \begin{array}{ll}
\bullet & E; (\{i\}; E; \{i\})^\omega \sqcap \{i\} \sqsubseteq (\{i\}; E; \{i\})^\omega \\
= & \{\text{Rule} : S \sqcap \{i\} = \{i\}; (S \sqcap \mathsf{skip})\} \\
 & \{i\}; (E; (\{i\}; E; \{i\})^\omega \sqcap \mathsf{skip}) \sqsubseteq (\{i\}; E; \{i\})^\omega \\
= & \{\text{Propagation of assertion}\} \\
 & \{i\}; (\{i\}; E; (\{i\}; E; \{i\})^\omega \sqcap \mathsf{skip}) \sqsubseteq (\{i\}; E; \{i\})^\omega \\
= & \{\text{Assumption (21)} : \{i\}; E = \{i\}; E; \{i\}\} \\
 & \{i\}; (\{i\}; E; \{i\}; (\{i\}; E; \{i\})^\omega \sqcap \mathsf{skip}) \sqsubseteq (\{i\}; E; \{i\})^\omega \\
= & \{\text{Unfolding [7]} : \ S^\omega = S; S^\omega \sqcap \mathsf{skip}\} \\
 & \{i\}; (\{i\}; E; \{i\})^\omega \sqsubseteq (\{i\}; E; \{i\})^\omega \\
\Leftarrow & \{\{i\} \sqsubseteq \mathsf{skip}\} \\
 & (\{i\}; E; \{i\})^\omega \sqsubseteq (\{i\}; E; \{i\})^\omega \\
= & \{\text{Reflexivity of } \sqsubseteq,\ S \sqsubseteq S\} \\
 & \top
\end{array} \\
 & (\{i\}; E; \{i\})^\omega
\end{array}$$

The proof in the other direction is straightforward using unfolding [7], $S^\omega = S; S^\omega \sqcap \mathsf{skip}$, and the property: $\{i\}; S \sqcap \mathsf{skip} = \{i\}; (S \sqcap \mathsf{skip})$. □

This means that we can propagate invariant assertions through loops and over other statements (see also (20) and (21)). These invariant assertions will not always be written out. However, they can be added anywhere in a program and to prove their introduction is straightforward although sometimes a bit tedious.

2.3 Data refinement

The underlying idea behind development in Event-B is the notion of *refinement.* An Event-B model can be developed stepwise, where each step introduces more features or more implementation details while preserving properties already proved in more abstract (simpler) models. A refinement Event-B model has the same structure as an abstract Event-B model. The only differences are that it contains a field to describe which model it refines and an *abstraction invariant* J that describes how the concrete and abstract state spaces relate. Here the refined Event-B model has variables w, which give a state space $\Gamma = \Gamma_1 \times \ldots \times \Gamma_m$. A refined event has the general form $E' \mathrel{\hat{=}} \textbf{when } H(w) \textbf{ then } w : |T(w, w') \textbf{ end}$. We then have the following set theoretic definitions [12]:

$$\begin{array}{lcl}
\Gamma & = & \{w | \top\} \\
k & = & \{v \mapsto w | I(v) \wedge J(v, w)\} \\
h & = & \{w | H(w)\} \\
t & = & \{w \mapsto w' | T(w, w')\} \\
b & = & \{w | B(w)\}
\end{array} \tag{22}$$

Here I is the invariant of the abstract model, J the abstraction invariant, H the guard of the concrete event E' and $w : |T(w, w')$ the substitution of event E'. The initialisation in the concrete model is given by the assignment of $w : |B$. To describe data refinement, a special decoding statement (set transformer) D is introduced that maps the concrete state space into the abstract state space.

For forward data refinement (normal Event-B refinement), we use a D such that $D(\phi) \mathrel{\hat{=}} k[\phi]$[2] [12]. Note that the only requirement in general is that D is monotonic [8]. However, the proof obligations generated by the Event-B tools are only valid if D has the form above. Note also that D is then non-miraculous (strict), $D(\mathsf{false}) = \mathsf{false}$. A statement S is data-refined by a statement S', $S \sqsubseteq_D S'$, iff [6, 8, 12]:

$$\forall s \cdot D; S(s) \subseteq S'; D(s) \tag{23}$$

Intuitively, this means that if S can reach a state in the subset s of the abstract state space then S' should reach the corresponding state in the concrete state space.

Assume we have an Event-B model $\mathcal{M}$ with variables v, initialisation *initialisation* and events E, as well as an concrete Event-B model $\mathcal{M}_1$ with variables w, initialisation *initialisation'*, events E' and E_n. We then have the following action system representations of the Event-B models $\mathcal{M}$ and $\mathcal{M}_1$:

$$\mathcal{M}_A \mathrel{\hat{=}} |[\ \mathbf{var}\ v;\ \mathbf{init}\ \mathit{initialisation};\ \mathbf{do}\ E\ \mathbf{od}\ \ ]| \tag{24}$$

$$\mathcal{M}_{1A} \mathrel{\hat{=}} |[\ \mathbf{var}\ w;\ \mathbf{init}\ \mathit{initialisation}';\ \mathbf{do}\ E' \sqcap E_n\ \mathbf{od}\ \ ]| \tag{25}$$

We have the definition that $\mathcal{M}$ is data refined by $\mathcal{M}_1$, $\mathcal{M} \sqsubseteq_D \mathcal{M}_1$, iff $\mathcal{M}_A \sqsubseteq_D \mathcal{M}_{1A}$. We then have that $\mathcal{M}_A \sqsubseteq_D \mathcal{M}_{1A}$ if the following conditions hold [7]:

$$D; \mathit{initialisation} \sqsubseteq \mathit{initialisation}'; D \tag{26}$$

$$D; E \sqsubseteq E'; D \tag{27}$$

$$D; \mathsf{skip} \sqsubseteq E_n; D \tag{28}$$

$$D; [\neg \mathsf{g}(E)] \sqsubseteq [\neg \mathsf{g}(E' \sqcap E_n)]; D \tag{29}$$

$$D(\mathsf{true}) \subseteq \mathbf{do}\ E_n\ \mathbf{od}\ (\mathsf{true}) \tag{30}$$

The first condition states that the abstract initialisation should be refined by the concrete one. Each old event E should then be refined by a corresponding event E'. Each new event E_n should refine skip. The forth condition states the concrete system should not terminate more often than the abstract system. The last condition requires that the new events terminate when executed in isolation. We have chosen to present this algebraic form of the data refinement rules, which can be easily used in the proofs presented here. All conditions above except for condition (29) are generated by the tools of Event-B [12].

To make the paper more self-contained a few useful refinement rules concerning assumptions and assertions are presented. These rules are frequently used later. The proofs are straightforward and can be found in e.g. [6]. Here g and h again denote predicates (sets):

$$\begin{array}{rcl@{\qquad\qquad}rcl} \{g\} \sqsubseteq \{h\} & \equiv & g \subseteq h & [g] \sqsubseteq [h] & \equiv & h \subseteq g \\ \{g\} & \sqsubseteq & \mathsf{skip} & \mathsf{skip} & \sqsubseteq & [g] \end{array} \tag{31}$$

3 Sequential program development in Event-B

Abrial has outlined [1, 2] a method for developing sequential programs using an event based approach. The idea is to first model the program as one event that performs the calculation in one step. The event non-deterministically assigns the variables a value satisfying the post-condition of the program. The parameters of the program are defined as constants and the pre-condition as axioms in a context of the Event-B model. Data refinement is then used to derive, in a stepwise manner, an algorithm satisfying the specification.

[2] $r[s]$ denotes the relational image of r from set s

3.1 Formal definitions

The idea [1, 2] can also be formalised in our action system setting. Since the result should be computed in one step according to the method for sequential program construction, there is no loop in the initial action system representation of the Event-B model. The initial specification consists of one event that is not inside a **do od** -loop:

$$\mathcal{E} \mathrel{\hat{=}} |[\ \textbf{var}\ v;\ \textbf{init}\ \mathit{initialisation}; E_0\ \ ]| \tag{32}$$

There is one event that calculates the result in one step, E_0. Here the precondition is given by the invariant and the constant properties (axioms) of the Event-B model. These properties are here carried in the invariant i. Note that due to (20), E_0 can be rewritten as $\{i\}; E_0$. This is similar to the program specification statement $\{p\}; [v := v'|S]$ [6], where p is the precondition and $[v := v'|S]$ denotes a non-deterministic assignment to the variables satisfying post-condition S. The specification model $\mathcal{E}$ is then refined into an event system $\mathcal{E}_1$ that implements the specification.

$$\mathcal{E}_1 \mathrel{\hat{=}} |[\ \textbf{var}\ w;\ \textbf{init}\ \mathit{initialisation}';\ \textbf{do}\ E_n\ \textbf{od}\ ; E_0'\ ]| \tag{33}$$

Here E_n denotes new events that describe the algorithm used to compute the desired result. The old event E_0' that described the post-condition has usually been reduced to only assigning the output variables the final result from the calculation.

Using the properties proved by the Event-B tools, the action systems with the semantics above are also correctly refined. We have proved:

$$D; \mathit{initialisation} \sqsubseteq \mathit{initialisation}'; D \tag{34}$$

$$D; E_0 \sqsubseteq E_0'; D \tag{35}$$

$$D; \mathsf{skip} \sqsubseteq \textbf{do}\ E_n\ \textbf{od}\ ; D \tag{36}$$

$$D; [\neg \mathsf{g}(E_0)] \sqsubseteq [\neg \mathsf{g}(E_0' \sqcap E_n)]; D \tag{37}$$

$$D(\mathsf{true}) \subseteq \textbf{do}\ E_n\ \textbf{od}\ (\mathsf{true}) \tag{38}$$

Note that we are only interested in the case when E_0 is non-miraculous, i.e., when the program is implementable. Using this fact, $\mathsf{g}(\{i\}; E_0) = \mathsf{true}$, then deadlock freeness condition (37) gives the condition:

$$D(\mathsf{true}) \subseteq \mathsf{g}(E_0' \sqcap E_n) \tag{39}$$

That event E_0 is non-miraculous can be ensured syntactically by not having a guard on the event [12].

Using the conditions above, we can now prove that $\mathcal{E} \sqsubseteq_D \mathcal{E}_1$ and that $\mathcal{E}_1$ is also non-miraculous. However, first we need a few additional properties. To prove that the refined system is non-miraculous, we need to take advantage of the invariant property stated in Lemma 2.

Lemma 2. $D; \{i\} \sqsubseteq \{D(i)\}; D$

Proof.

$$\begin{array}{ll}
 & D; \{i\} \sqsubseteq \{D(i)\}; D \\
\equiv & \{\text{Definition}\} \\
 & \forall s \cdot D; \{i\}(s) \subseteq \{D(i)\}; D(s) \\
\equiv & \{\text{wp} - \text{calculations}:\ (3)\ \text{and}\ (6)\} \\
 & \forall s \cdot D(i \cap s) \subseteq D(i) \cap D(s) \\
\equiv & \{i \cap s \subseteq i\ \text{and}\ i \cap s \subseteq s\ \text{and}\ D\ \text{monotonic}\} \\
 & \top
\end{array}$$

□

We also need to prove that invariant assertions can be introduced in the refined specification. This means that the abstract invariant should be established by the concrete initialisation *initialisation′* and maintained by each event $E \in E' \cup E_n$ in the concrete model:

$$initialisation' = initialisation'; \{D(i)\} \tag{40}$$

$$\{D(i)\}; E = \{D(i)\}; E; \{D(i)\} \tag{41}$$

This can be proved from the invariant preservation of abstract events and the refinement conditions. The proof is omitted here for brevity. See e.g. [8] (Theorem 34) for how the proof can be carried out. This means that the assertion $\{D(i)\}$ can be used in the same manner as the invariant assertion $\{i\}$ in the abstract specification.

Lemma 3. *If* $\mathsf{g}(\{i\}; E_0) = \mathsf{true}$ *and the refinement proof obligations (34)-(38) has been discharged then* $\mathcal{E} \sqsubseteq_D \mathcal{E}_1$ *and* $\mathcal{E}_1$ *will not behave miraculously.*

Proof. Refinement of the events gives:

$$\begin{array}{ll} & D; E_0 \\ \sqsubseteq & \{E_0 = \mathsf{skip}; E_0 \text{ and } (36) : D; \mathsf{skip} \sqsubseteq \textbf{do } E_n \textbf{ od }; D\} \\ & \textbf{do } E_n \textbf{ od }; D; E_0 \\ \sqsubseteq & \{D; E_0 \sqsubseteq E_0'; D\} \\ & \textbf{do } E_n \textbf{ od }; E_0'; D \end{array}$$

Then we prove that the refinement is non-miraculous, i.e. $\mathsf{g}(S) = \mathsf{true}$:

$$\begin{array}{ll} & \textbf{do } E_n \textbf{ od }; E_0'; D \ (\mathsf{false}) \\ = & \{\text{Property of } D\text{: } D(\mathsf{false}) = \mathsf{false}\} \\ & \textbf{do } E_n \textbf{ od }; E_0' \ (\mathsf{false}) \\ = & \{\text{Definition of } \textbf{do od }\} \\ & E_n^\omega; [\neg \mathsf{g}(E_n)]; E_0' \ (\mathsf{false}) \\ = & \{\text{Rule :} [g] = [g]; \{g\}\} \\ & E_n^\omega; [\neg \mathsf{g}(E_n)]; \{\neg \mathsf{g}(E_n)\}; E_0' \ (\mathsf{false}) \\ = & \{\text{Invariant introduction}\} \\ & E_n^\omega; [\neg \mathsf{g}(E_n)]; \{D(i)\}; \{\neg \mathsf{g}(E_n)\}; E_0' \ (\mathsf{false}) \\ \subseteq & \{i \subseteq \mathsf{true} \text{ and } D(i) \subseteq D(\mathsf{true}) \text{ since } D \text{ monotonic,} \\ & \quad \text{Rule : } g \subseteq h \equiv \{g\} \sqsubseteq \{h\}\} \\ & E_n^\omega; [\neg \mathsf{g}(E_n)]; \{D(\mathsf{true})\}; \{\neg \mathsf{g}(E_n)\}; E_0' \ (\mathsf{false}) \\ \subseteq & \{\text{Assumption (39) : } \ D(\mathsf{true}) \subseteq \mathsf{g}(E_0') \cup \mathsf{g}(E_n)\} \\ & E_n^\omega; [\neg \mathsf{g}(E_n)]; \{\mathsf{g}(E_0') \cup \mathsf{g}(E_n)\}; \{\neg \mathsf{g}(E_n')\}; E_0 \ (\mathsf{false}) \\ = & \{\{g\}; \{h\} = \{g \cap h\} \text{ and set theory}\} \\ & E_n^\omega; [\neg \mathsf{g}(E_n)]; \{\neg \mathsf{g}(E_n) \cap \mathsf{g}(E_0')\}; E_0' \ (\mathsf{false}) \\ = & \{\text{Rule : } \ \{g \cap h\} = \{g\}; \{h\}\} \\ & E_n^\omega; [\neg \mathsf{g}(E_n)]; \{\neg \mathsf{g}(E_n)\}; \{\mathsf{g}(E_0')\}; E_0' \ (\mathsf{false}) \\ = & \{\text{Rule : } \ \{\mathsf{g}(S)\}; S \ (\mathsf{false}) = \mathsf{false}\} \\ & E_n^\omega; [\neg \mathsf{g}(E_n)]; \{\neg \mathsf{g}(E_n)\} \ (\mathsf{false}) \\ = & \{\text{Rule : } \ \{g\} \ (\mathsf{false}) = \mathsf{false}\} \\ & E_n^\omega; [\neg \mathsf{g}(E_n)] \ (\mathsf{false}) \\ = & \{\text{Rule (Lemma 17b in [7]) : } S^\omega; [\neg \mathsf{g}(S)](\mathsf{false}) = \mathsf{false}\} \\ & \mathsf{false} \end{array}$$

□

In [12] introduction of while-loops is done in a similar way. However, here we are interested in implementable specifications and consequently the abstract event needs to be non-miraculous. Hallerstede assumes introduction of loops in a more general setting where the abstract event can have a guard and that the guard is interpreted as the pre-condition of the complete loop. The proof obligation required in that setting is exactly the traditional deadlock freeness proof obligation in (37). We have a slightly stronger version where the abstract event is assumed to be always enabled and we also only prove that deadlock freeness proof obligation is sufficient. The proof in [12] could, of course, also be reused here.

3.2 Example

To illustrate the development approach, a small example involving a program for summing the elements in an array is presented. The array with the elements to sum is denoted by f and the size of the array is given by m. The result is given by the variable *sum*.

```
context  $\mathcal{C}$
constants  $m, f$
axioms
    $m \in \mathbb{N}_1$
    $f \in 1..m \rightarrow \mathbb{Z}$
end
```

```
machine  $\mathcal{M}$
sees  $\mathcal{C}$
variables  $sum$
invariant
  $sum \in \mathbb{Z}$
events
$initialisation \mathrel{\hat{=}}$
begin
  $sum := 0$
end
$calc \mathrel{\hat{=}}$
begin
  $sum := \Sigma_{i=1}^{m} f(i)$
end
end
```

In the abstract specification the computation is performed in one step, $sum := \Sigma_{i=1}^{m} f(i)$. The event thus encodes that the variables are assigned values satisfying the post-condition of the computation. In the refinement, a new event *progress* is introduced to iteratively calculate the result. Here s holds the intermediate result and j is the end of the sub-array summed so far. We get the following model:

```
machine  $\mathcal{M}_1$
refines  $\mathcal{M}$
sees  $\mathcal{C}$
variables  $sum, j, s$
invariant
    $s \in \mathbb{Z}$
    $j \in 0..m$
    $j > 0 \Rightarrow (s = \Sigma_{i=1}^{j} f(i))$
events
$initialisation \mathrel{\hat{=}}$
begin
    $sum := 0$
    $s := 0$
    $j := 0$
end
```

```
$progress \mathrel{\hat{=}}$
where $j < m$
then
    $j := j + 1$
    $s := s + f(j + 1)$
end
$calc \mathrel{\hat{=}}$
where $j = m$
then $sum := s$
end
end
```

The models $\mathcal{M}$ and $\mathcal{M}_1$ above give the action system representations:

$$\mathcal{M}_A \mathrel{\hat{=}} |[\ \mathbf{var}\ sum;\ \mathbf{init}\ [initialisation];\ [calc]\ \ ]| \tag{42}$$

$$\begin{aligned}&\mathcal{M}_{1A} \mathrel{\hat{=}}\\ &\quad |[\ \mathbf{var}\ sum, j, s;\ \mathbf{init}\ [initialisation];\ \mathbf{do}\ [progress]\ \mathbf{od}\ ; [calc]\ \ ]|\end{aligned} \tag{43}$$

According to the patterns for mapping events to imperative programming constructs [1, 2], the model $\mathcal{M}_1$ would be implemented as following statement:

```
$sum := 0$;
$s := 0$;
$j := 0$;
while $j < m$
then
    $j := j + 1$
    $s := s + f(j + 1)$
end ;
$sum := s$
```

The patterns in [1, 2] give a convenient way to implement sequential programs from Event-B models. However, the patterns do not handle general sequential composition. For example, consider the example above where it is known that $m = 3$ and we would like to implement the summation as an unrolled loop. Loop unrolling is a common optimisation technique applied by hand or automatically by the compiler to make loops that are iterated only a few times run faster. We can further refine $\mathcal{M}_1$ to do the unrolling:

```
machine  $\mathcal{M}_2$
refines  $\mathcal{M}_1$
sees  $\mathcal{C}$
variables  $sum, j, s$
invariant
    $s \in \mathbb{Z}$
    $j \in 0..m$
    $j > 0 \Rightarrow (s = \Sigma_{i=1}^{j} f(i))$
events
$initialisation \hat{=}$
begin
    $sum := 0$
    $s := 0$
    $j := 0$
end
```

```
$progress_1 \hat{=}$
where $j = 0$
then
    $j := 1$
    $s := s + f(1)$
end
$progress_2 \hat{=}$
where $j = 1$
then
    $j := 2$
    $s := s + f(2)$
end
$progress_3 \hat{=}$
where $j = 2$
then
    $j := 3$
    $s := s + f(3)$
end
$calc \hat{=}$
where $j = 3$
then $sum := s$
end
end
```

The desired implementation of this model is:

```
$sum := 0$;
$s := 0$;
$s := s + f(1)$;  $s := s + f(2)$;  $s := s + f(3)$
$sum := s$
```

This implementation cannot be obtained by the implementation patterns in [1, 2]. In the following sections we develop a method where it is possible to prove that the implementation above refines the Event-B model $\mathcal{M}_2$.

4 Creation of sequential program statements by scheduling events

To create efficient sequential programs, we like to introduce more precise flow of control. This means that we create a schedule for the events which they should be executed according to. In order for the schedule to be correct, we have to show that executing the events according to it leads to a refinement of the old loop of events. First we describe a scheduling language. Based on this language, the events from the Event-B model are scheduled to obtain a sequential program statement. We then show how to prove the correctness of this statement. Then we present a few reusable patterns for scheduling, as well as a few patterns to refine the statement obtained by scheduling events into an imperative program.

4.1 The scheduling language

First we introduce a scheduling language to describe the scheduling of events. It has the following grammar:

$$\begin{aligned}
&\mathit{DoStmnt} ::= \ [\{g\} \rightarrow]\ \textbf{do}\ \mathit{Stmnt}\ \textbf{od} \\
&\mathit{ChoiceStmnt} ::= \ E\ (\|\ \ E)^* \\
&\mathit{Stmnt} ::= \ (\mathit{DoStmnt} \mid \mathit{ChoiceStmnt})(\rightarrow\ \ \mathit{DoStmnt} \mid \mathit{ChoiceStmnt})^* \\
&\qquad\qquad \rightarrow \mathsf{hlt}
\end{aligned} \tag{44}$$

Here exp^* denotes zero or more occurrences of exp, $[exp]$ optional occurrence of exp, g is a predicate and E is the name of an event. We can have sequential composition of events $\rightarrow$, choice of events $\|$ and iteration of events **do od** . We can also have assertions $\{g\}$ before a loop. This is often needed for the proofs of loop correctness. The label hlt denotes the end of a statement. It could be omitted, but it is used here to make the scheduling rules presented later simpler. The language is not very flexible. However, the aim is that it should be possible to automatically generate proof obligations for schedule satisfaction, as well as easily map the statement obtained after scheduling to an imperative programming language. It is possible to calculate the necessary proof obligations from the guards and the assertions given in the schedule. Extra events or extra variables are not necessary for verification. There are usually many ways to show that a schedule can be satisfied and general rules that are usable are therefore difficult to obtain. Often specialised proof strategies that take advantage of model-specific properties about events are needed. To remedy this problem this language can easily be used to develop and verify reusable patterns for scheduling with known verification conditions.

The correctness criteria for the statement obtained by scheduling the events is that it refines the original loop of events. The scheduled statement should also be non-miraculous. The verification of the scheduling is here done incrementally in a top-down manner. Assume that we have a recursive decent parser sched of schedules that conform to the grammar in (44), which also acts as one-pass compiler from schedule to sequential statement. We show that each recursive call to sched will create a refinement of the previous call. Due to monotonicity of all set transformers involved, this will ensure that at the end the final statement obtained from the schedule refines the original Event-B model. This type of stepwise verification of scheduling seems to be well suited for creation and verification of reusable patterns for introducing different types of flow control. This is not the only option for scheduling. The creation of sequential programs in [1, 2] is done in a more bottom-up manner, where individual events are merged to form a larger units. There are often side-conditions for the scheduling patterns stating that the rest of the system does not interfere. In our approach those side conditions are explicitly taken into account, which might be more difficult to do in a bottom-up approach.

4.2 Verification of Scheduling

Let the schedule compiler function $\mathsf{sched}(E, S)$ be a function from a set of events E and a schedule S to the statement obtained after scheduling of events according to the schedule. The events E are here the events in schedule S. Below E, E_1 and E_2 denote sets of events. E.g. the set E_1 is assumed to consist of events $E_{11}, \ldots, E_{1n}$ and thus E_1 in the schedule denotes $E_{11} \parallel \ldots \parallel E_{1n}$. Here we use the notation $[E_1]$ to mean the demonic choice of all events in E_1, $[E_1] \mathrel{\hat{=}} [E_{11}] \sqcap \ldots \sqcap [E_{1n}]$. The function sched is recursively defined as follows:

$$\begin{array}{l}
\mathsf{sched}(E \cup E_1 \cup E_2,\ E_1 \to E_2 \to S) \longrightarrow \\
\quad [E_1]; \{\mathsf{g}([E_2])\}; \mathsf{sched}(E \cup E_2,\ E_2 \to S) \\
\mathsf{sched}(E \cup E_1 \cup E_2,\ E_1 \to \{g\} \to \textbf{do } S_2 \textbf{ od } \to S) \longrightarrow \\
\quad [E_1]; \{g\}; \mathsf{sched}(E \cup E_2,\ \{g\} \to \textbf{do } S_2 \textbf{ od } \to S) \\
\mathsf{sched}(E \cup E_1 \cup E_2, \{g\} \to \textbf{do } S_1 \textbf{ od } \to E_2 \to S) \longrightarrow \\
\quad \{g\}; ([\mathsf{g}([E_1])]; \mathsf{sched}(E_1, S_1))^\omega; [\neg\mathsf{g}(E_1)]; \{\mathsf{g}(E_2)\}; \\
\quad \mathsf{sched}(E \cup E_1,\ E_2 \to S) \\
\mathsf{sched}(E \cup E_1 \cup E_2,\ \{g_1\} \to \textbf{do } S_1 \textbf{ od } \to \{g_2\} \to \textbf{do } E_2 \textbf{ od } \to S) \longrightarrow \\
\quad \{g_1\}; ([\mathsf{g}([E_1])]; \mathsf{sched}(E_1, S_1))^\omega; [\neg\mathsf{g}([E_1])]; \{g_2\}; \\
\quad \mathsf{sched}(E \cup E_1,\ \{g_2\} \to \textbf{do } E_2 \textbf{ od } \to S) \\
\mathsf{sched}(E_1,\ E_1 \to \mathsf{hlt}) \longrightarrow \\
\quad [E_1] \\
\mathsf{sched}(E_1,\ \{g\} \to \textbf{do } S_1 \textbf{ od } \to \mathsf{hlt}) \longrightarrow \\
\quad \{g\}; ([\mathsf{g}([E_1])]; \mathsf{sched}(E_1, S_1))^\omega; [\neg\mathsf{g}([E_1])]
\end{array} \tag{45}$$

To emphasize the direction of function application, we have used $\longrightarrow$ instead of $=$ for definition of the function sched. We have also here omitted the case when the assertion $\{g\}$ is not present before the loop, since it can be handled using the identity $\{\mathsf{true}\} = \mathsf{skip}$. To verify that the events can be scheduled according to the desired schedule, each application of a scheduling function should lead to a refinement of the previous step. The function application $\mathsf{sched}(E, S)$ above thus refers to events that have not been scheduled yet and its semantics is therefore $[E]^\omega; [\neg\mathsf{g}([E])]$. Hence, we prove that the left-hand side is always refined by the right hand side. Furthermore, it might also be desirable to prove that the right hand side is non-miraculous. For simplicity, from here on we directly denote the set transformer $[E]$ corresponding to the event names E with only E.

Note that on the right hand side, scheduling statements of the form $\mathsf{sched}(E \cup E_i, S_i \to S)$ are always preceded by a assertion that ensures that the following statement in the schedule is enabled and that it terminates. This means that we can usually do the desired refinement proofs in a *context* [6] where a context assertion holds. In the proof obligation below we have assumed that all applications of sched in (45) were done in a context, where the assertion $\{c\}$ holds. We then have refinement conditions of the form $\{c\}; \mathsf{sched}(E \cup E_i, S_i \to S) \sqsubseteq \ldots$.

The proof obligations for the scheduling using sched from (45) become:

$$\begin{array}{l}
\{c\}; (E \sqcap E_1 \sqcap E_2)^\omega; [\neg\mathsf{g}(E \sqcap E_1 \sqcap E_2)] \sqsubseteq \\
E_1; \{\mathsf{g}(E_2)\}; (E \sqcap E_2)^\omega; [\neg\mathsf{g}(E \sqcap E_2)]
\end{array} \tag{46}$$

$$\begin{array}{l}
\{c\}; (E \sqcap E_1 \sqcap E_2)^\omega; [\neg\mathsf{g}(E \sqcap E_1 \sqcap E_2)] \sqsubseteq \\
E_1; \{g\}; (E \sqcap E_2)^\omega; [\neg\mathsf{g}(E \sqcap E_2)]
\end{array} \tag{47}$$

$$\begin{array}{l}
\{c\}; (E \sqcap E_1 \sqcap E_2)^\omega; [\neg\mathsf{g}(E \sqcap E_1 \sqcap E_2)] \sqsubseteq \\
\{g\}; ([\mathsf{g}(E_1)]; \mathsf{sched}(E_1, S_1))^\omega; [\neg\mathsf{g}(E_1)]; \{\mathsf{g}(E_2)\}; (E \sqcap E_2)^\omega; [\neg\mathsf{g}(E \sqcap E_2)]
\end{array} \tag{48}$$

$$\begin{aligned}&\{c\}; (E \sqcap E_1 \sqcap E_2)^\omega; [\neg \mathsf{g}(E \sqcap E_1 \sqcap E_2)] \sqsubseteq \\ &\{g_1\}; ([\mathsf{g}(E_1)]; \mathsf{sched}(E_1, S_1))^\omega; [\neg \mathsf{g}(E_1)]; \{g_2\}; (E \sqcap E_2)^\omega; [\neg \mathsf{g}(E \sqcap E_2)]\end{aligned} \tag{49}$$

$$\{c\}; E_1^\omega; [\neg \mathsf{g}(E_1)] \sqsubseteq E_1 \tag{50}$$

$$\{c\}; E_1^\omega; [\neg \mathsf{g}(E_1)] \sqsubseteq \{g\}; ([\mathsf{g}(E_1)]; \mathsf{sched}(E_1, S_1))^\omega; [\neg \mathsf{g}(E_1)] \tag{51}$$

The function application $\mathsf{sched}(E_1, S_1)$ is interpreted as a still unscheduled loop and thus it is equal to $E_1^\omega; [\neg \mathsf{g}(E_1)]$. Lemma 4 can then be used to simplify the statement $([\mathsf{g}(E_1)]; \mathsf{sched}(E_1, S_1))^\omega$, which has the semantics $([\mathsf{g}(E_1)]; E_1^\omega; [\neg \mathsf{g}(E_1)])^\omega; [\neg \mathsf{g}(E_1)]$.

Lemma 4. $S^\omega; [\neg \mathsf{g}(S)] = ([\mathsf{g}(S)]; S^\omega; [\neg \mathsf{g}(S)])^\omega; [\neg \mathsf{g}(S)]$

Proof.

$$\begin{aligned}
& ([\mathsf{g}(S)]; S^\omega; [\neg \mathsf{g}(S)])^\omega; [\neg \mathsf{g}(S)] \\
= \; & \{\text{Unfolding [7]}: \; S^\omega = S; S^\omega \sqcap \mathsf{skip}\} \\
& ([\mathsf{g}(S)]; S^\omega; [\neg \mathsf{g}(S)]; ([\mathsf{g}(S)]; S^\omega; [\neg \mathsf{g}(S)])^\omega \sqcap \mathsf{skip}); [\neg \mathsf{g}(S)] \\
= \; & \{\text{Leapfrog (Lemma 11 in [7])}: S; (T; S)^\omega = (S; T)^\omega; S\} \\
& ([\mathsf{g}(S)]; S^\omega; ([\neg \mathsf{g}(S)]; [\mathsf{g}(S)]; S^\omega)^\omega; [\neg \mathsf{g}(S)] \sqcap \mathsf{skip}); [\neg \mathsf{g}(S)] \\
= \; & \{[\neg \mathsf{g}(S)]; [\mathsf{g}(S)] = \mathsf{magic} \text{ and } \mathsf{magic}; S = \mathsf{magic} \text{ and } \mathsf{magic}^\omega = \mathsf{skip}\} \\
& ([\mathsf{g}(S)]; S^\omega; [\neg \mathsf{g}(S)] \sqcap \mathsf{skip}); [\neg \mathsf{g}(S)] \\
= \; & \{\text{Unfolding [7]}: \; S^\omega = S; S^\omega \sqcap \mathsf{skip}\} \\
& ([\mathsf{g}(S)]; (S; S^\omega \sqcap \mathsf{skip}); [\neg \mathsf{g}(S)] \sqcap \mathsf{skip}); [\neg \mathsf{g}(S)] \\
= \; & \{\text{Distribution over } \sqcap \text{ and } [\mathsf{g}(S)]; [\neg \mathsf{g}(S)] = \mathsf{magic}\} \\
& (([\mathsf{g}(S)]; S; S^\omega; [\neg \mathsf{g}(S)] \sqcap \mathsf{magic}) \sqcap \mathsf{skip}); [\neg \mathsf{g}(S)] \\
= \; & \{\text{Definition of } \mathsf{g} \text{ and } S \sqcap \mathsf{magic} = S\} \\
& (S; S^\omega; [\neg \mathsf{g}(S)] \sqcap \mathsf{skip}); [\neg \mathsf{g}(S)] \\
= \; & \{\text{Distribution over } \sqcap \text{ and rule}: \; [g]; [g] = [g]\} \\
& (S; S^\omega \sqcap \mathsf{skip}); [\neg \mathsf{g}(S)] \\
= \; & \{\text{Unfolding [7]}: \; S^\omega = S; S^\omega \sqcap \mathsf{skip}\} \\
& S^\omega; [\neg \mathsf{g}(S)]
\end{aligned}$$

□

4.3 Scheduling patterns

The proof obligations in (46)-(51) cannot be proved for arbitrary events. They can only be proved when the events involved satisfy certain constraints. There are also many possibilities for the schedules to actually be satisfied. The best approach to scheduling is probably to develop a set of patterns with known correctness conditions. The key to the development of patterns is that we can prove incrementally that each application of the scheduling function sched result in a non-miraculous refinement. This can be used to show that a certain sequence of scheduling steps (i.e. a pattern) results in a refinement even though the Event-B model has not been fully scheduled. A pattern P consists of a *pre-condition* for the pattern, a *schedule* describing the pattern and a *list of assumptions* about the events in the schedule. The pre-condition states under which conditions the pattern can be applied. It can thus be used as a *context assertion* when proving the correctness of the pattern. The schedule describes the scheduling statement the pattern concerns. The list of assumptions describes the assumptions about the events that have to hold before the pattern can be applied.

Loop introduction Here we will first prove the correctness of the pattern for loop introduction in [1, 2]. The pattern states that a loop of events with non-deterministic choice can be refined into a loop where the first event is iterated until it becomes disabled and the second event is then executed. The goal is to introduce an inner while-loop so that the Event-B model to the left below is refined by the one to the right.

$$\boxed{\begin{array}{l} E_1 \mathrel{\hat=} \textbf{when } G_1 \textbf{ then } S_1 \textbf{ end} \\ E_2 \mathrel{\hat=} \textbf{when } G_2 \textbf{ then } S_2 \textbf{ end} \end{array}} \quad \sqsubseteq \quad \boxed{\begin{array}{l} E \mathrel{\hat=} \\ \textbf{when } G_1 \vee G_2 \textbf{ then} \\ \quad \textbf{while } G_1 \textbf{ then } S_1 \textbf{ end } ; \\ \quad S_2 \\ \textbf{end} \end{array}}$$

The pattern above can be described as the scheduling pattern P_1 in (52), which has the parameters E_1 and E_2 denoting two sets of events, as well as S_1 denoting an arbitrary schedule. Here we do the generalisation of the pattern in [1, 2] that we assume that the events E_1 in the inner loop are scheduled according to some unknown schedule S_1.

$$\begin{array}{lcl} \multicolumn{3}{l}{P_1(E_1, E_2, S_1) \mathrel{\hat=}} \\ \quad \text{Precondition} & : & \mathsf{true} \\ \quad \text{Schedule} & : & \textbf{do do } S_1 \rightarrow \mathsf{hlt} \textbf{ od } \rightarrow E_2 \textbf{ od } \rightarrow \mathsf{hlt} \\ \quad \text{Assumption 1} & : & \{i \cap \mathsf{g}(E_1 \sqcap E_2)\}; E_1 = \\ & & \{i \cap \mathsf{g}(E_1 \sqcap E_2)\}; E_1; \{\mathsf{g}(E_1 \sqcap E_2)\} \end{array} \tag{52}$$

Application of function sched results in the statement:

$$\begin{array}{l} \mathsf{sched}(E_1 \cup E_2, \textbf{do } S_1 \rightarrow \mathsf{hlt} \textbf{ od } \rightarrow E_2 \rightarrow \mathsf{hlt}) \longrightarrow \\ ([\mathsf{g}(E_1 \sqcap E_2)]; ([\mathsf{g}(E_1)]; \mathsf{sched}(E_1, S_1))^\omega; [\neg\mathsf{g}(E_1)]; \{\mathsf{g}(E_2)\}; E_2)^\omega; [\neg\mathsf{g}(E_1 \sqcap E_2)] \end{array}$$

Note that we still have one un-scheduled part $\mathsf{sched}(E_1, S_1)$, which is unknown. As before, this statement is interpreted as $E_1^\omega; [\neg\mathsf{g}(E_1)]$ at this level. We get the following condition to prove:

$$\begin{array}{l} (E_1 \sqcap E_2)^\omega; [\neg\mathsf{g}(E_1 \sqcap E_2)] \\ \sqsubseteq \\ ([\mathsf{g}(E_1 \sqcap E_2)]; ([\mathsf{g}(E_1)]; E_1^\omega; [\neg\mathsf{g}(E_1)])^\omega; [\neg\mathsf{g}(E_1)]; \{\mathsf{g}(E_2)\}; E_2)^\omega; [\neg\mathsf{g}(E_1 \sqcap E_2)] \end{array}$$

This refinement condition does not hold in general and Assumption 1 about the events is needed. This assumption states that E_1 does not disable both events. This is the case if E_1 was introduced after E_2 in the refinement chain and the value of $\mathsf{g}(E_1 \sqcap E_2)$ depends only on variables introduced before or simultaneously with E_2 (E_1 cannot change the value of this condition then). This requirement is also discussed in [2]. The refinement can now be proved. Note again that in all the proofs here, we do not write out invariant assertions. They can, however, be added between each statement.

Proof.

$$
\begin{array}{ll}
 & (E_1 \sqcap E_2)^\omega ; [\neg \mathsf{g}(E_1 \sqcap E_2)] \\
= & \{\text{Decomposition (Lemma 12 in [7])}: \ (S \sqcap T)^\omega = S^\omega ; (T ; S^\omega)^\omega\} \\
 & E_1^\omega ; (E_2 ; E_1^\omega)^\omega ; [\neg \mathsf{g}(E_1 \sqcap E_2)] \\
= & \{\text{Leapfrog (Lemma 11 in [7])}: \ S ; (T ; S)^\omega = (S ; T)^\omega ; S\} \\
 & (E_1^\omega ; E_2)^\omega ; E_1^\omega ; [\neg \mathsf{g}(E_1 \sqcap E_2)] \\
\sqsubseteq & \{\text{Rule}: \ S^\omega \sqsubseteq \mathsf{skip}\} \\
 & (E_1^\omega ; E_2)^\omega ; [\neg \mathsf{g}(E_1 \sqcap E_2)] \\
\sqsubseteq & \{\text{Assumption introduction}: \ \mathsf{skip} \sqsubseteq [g]\} \\
 & (E_1^\omega ; [\neg \mathsf{g}(E_1)] ; E_2)^\omega ; [\neg \mathsf{g}(E_1 \sqcap E_2)] \\
= & \{\mathsf{g}(E_1) \subseteq \mathsf{g}(E_1 \sqcap E_2) \text{ and } g \subseteq h \Rightarrow [g] = [h] ; [g]\} \\
 & ([\mathsf{g}(E_1 \sqcap E_2)] ; E_1^\omega ; [\neg \mathsf{g}(E_1)] ; E_2)^\omega ; [\neg \mathsf{g}(E_1 \sqcap E_2)] \\
= & \{\text{Rule}: \ [g] = [g] ; \{g\}\} \\
 & ([\mathsf{g}(E_1 \sqcap E_2)] ; \{\mathsf{g}(E_1 \sqcap E_2)\} ; E_1^\omega ; [\neg \mathsf{g}(E_1)] ; E_2)^\omega ; [\neg \mathsf{g}(E_1 \sqcap E_2)] \\
\sqsubseteq & \{\text{Assumption 1 and Lemma 14c in [7]}: \ S ; T \sqsubseteq U ; S \Rightarrow S ; T^\omega \sqsubseteq U^\omega ; S\} \\
 & ([\mathsf{g}(E_1 \sqcap E_2)] ; E_1^\omega ; \{\mathsf{g}(E_1 \sqcap E_2)\} ; [\neg \mathsf{g}(E_1)] ; E_2)^\omega ; [\neg \mathsf{g}(E_1 \sqcap E_2)] \\
\sqsubseteq & \{\text{Distribution of assertions over assumptions and rule}: \ [g] = [g]\{g\}\} \\
 & ([\mathsf{g}(E_1 \sqcap E_2)] ; E_1^\omega ; [\neg \mathsf{g}(E_1)] ; \{\neg \mathsf{g}(E_1)\} ; \{\mathsf{g}(E_1 \sqcap E_2)\} ; E_2)^\omega ; [\neg \mathsf{g}(E_1 \sqcap E_2)] \\
= & \{\text{Assertion properties}: \ \{g\} ; \{h\} = \{g \cap h\} \text{ and definition of } \mathsf{g}\} \\
 & ([\mathsf{g}(E_1 \sqcap E_2)] ; E_1^\omega ; [\neg \mathsf{g}(E_1)] ; \{\neg \mathsf{g}(E_1) \cap (\mathsf{g}(E_1) \cup \mathsf{g}(E_2))\} ; E_2)^\omega ; \\
 & \quad [\neg \mathsf{g}(E_1 \sqcap E_2)] \\
= & \{\text{Set theory}\} \\
 & ([\mathsf{g}(E_1 \sqcap E_2)] ; E_1^\omega ; [\neg \mathsf{g}(E_1)] ; \{\neg \mathsf{g}(E_1) \cap \mathsf{g}(E_2)\} ; E_2)^\omega ; [\neg \mathsf{g}(E_1 \sqcap E_2)] \\
= & \{\text{Rule}: \ \{g \cap h\} = \{g\} ; \{h\} \text{ and } [g] = [g] ; \{g\}\} \\
 & ([\mathsf{g}(E_1 \sqcap E_2)] ; E_1^\omega ; [\neg \mathsf{g}(E_1)] ; \{\mathsf{g}(E_2)\} ; E_2)^\omega ; [\neg \mathsf{g}(E_1 \sqcap E_2)] \\
= & \{\text{Lemma 4}\} \\
 & ([\mathsf{g}(E_1 \sqcap E_2)] ; ([\mathsf{g}(E_1)] ; E_1^\omega ; [\neg \mathsf{g}(E_1)])^\omega ; [\neg \mathsf{g}(E_1)] ; \{\mathsf{g}(E_2)\} ; E_2)^\omega ; \\
 & \quad [\neg \mathsf{g}(E_1 \sqcap E_2)]
\end{array}
$$

In order to ensure that the refined statement is non-miraculous, we prove that $\mathsf{g}([\mathsf{g}(E_1 \sqcap E_2)] ; ([\mathsf{g}(E_1)] ; E_1)^\omega ; [\neg \mathsf{g}(E_1)] ; E_2) = \mathsf{g}(E_1 \sqcap E_2)$ and then use Lemma 17b in [7], $S^\omega ; [\neg \mathsf{g}(S)](\mathsf{false}) = \mathsf{false}$.

$$
\begin{array}{ll}
 & \mathsf{g}([\mathsf{g}(E_1 \sqcap E_2)] ; ([\mathsf{g}(E_1)] ; E_1^\omega ; [\neg \mathsf{g}(E_1)])^\omega ; [\neg \mathsf{g}(E_1)] ; \{\mathsf{g}(E_2)\} ; E_2) \\
= & \{\text{Lemma 4}\} \\
 & \mathsf{g}([\mathsf{g}(E_1 \sqcap E_2)] ; E_1^\omega ; [\neg \mathsf{g}(E_1)] ; \{\mathsf{g}(E_2)\} ; E_2) \\
= & \{\text{Definition of } \mathsf{g}\} \\
 & \neg [\mathsf{g}(E_1 \sqcap E_2)] ; E_1^\omega ; [\neg \mathsf{g}(E_1)] ; \{\mathsf{g}(E_2)\} ; E_2 \ (\mathsf{false}) \\
= & \{\{\mathsf{g}(E_2)\} ; E_2 \ (\mathsf{false}) = \mathsf{false}\} \\
 & \neg [\mathsf{g}(E_1 \sqcap E_2)] ; E_1^\omega ; [\neg \mathsf{g}(E_1)] \ (\mathsf{false}) \\
= & \{\text{Rule (Lemma 17b in [7])}: S^\omega ; [\neg \mathsf{g}(S)](\mathsf{false}) = \mathsf{false}\} \\
 & \neg [\mathsf{g}(E_1 \sqcap E_2)] \ (\mathsf{false}) \\
= & \{\text{Definitions}\} \\
 & \neg(\neg \mathsf{g}(E_1 \sqcap E_2) \cup \mathsf{false}) \\
= & \{\text{Set theory}\} \\
 & \mathsf{g}(E_1 \sqcap E_2)
\end{array}
$$

□

This demonstrates one possible proof of one simple schedule. Note that the user of this scheduling pattern will only have to prove Assumption 1.

Sequential composition One of the most important types of patterns concerns the introduction of sequential composition of events. These patterns are already given as the equations one, two and five in the definition of sched (45). They are thus patterns that concern only one application of sched. Here we focus on the pattern from equation one:

$$\begin{array}{lll} P_{seq}(E_1, E_2, S') \mathrel{\hat{=}} & & \\ \quad \text{Precondition} & : & \mathsf{g}(E_1) \\ \quad \text{Schedule} & : & E_1 \to (E_2 \to S') \\ \quad \text{Assumption 1} & : & \{i \cap \neg\mathsf{g}(E_1)\}; (E_2 \sqcap E) = \\ & & \{i \cap \neg\mathsf{g}(E_1)\}; (E_2 \sqcap E); \{\neg\mathsf{g}(E_1)\} \\ \quad \text{Assumption 2} & : & \{i\}; E_1 = \{i\}; E_1; \{\neg\mathsf{g}(E_1) \cap \mathsf{g}(E_2)\} \end{array} \tag{53}$$

To verify this pattern, we have to prove condition (46). This condition cannot be proved directly and we therefore need the additional assumptions about the events. There are also no unique assumptions that would enable the proofs, but there are several possibilities. We here show how to prove correctness of the scheduling based on the strategy that event E_1 occurs only once in a schedule and once it has become disabled it remains disabled. This property is stated in Assumption 1, where E denotes the events in S'. This assumption is not sufficient. To discharge the proof obligation, Assumption 2 is also needed. The assumption states that E_1 should disable itself and enable the event E_2 after it. We also use the context assertion $\{\mathsf{g}(E_1)\}$ from the pre-condition of the pattern.

Proof.

$$\begin{array}{ll} & \{\mathsf{g}(E_1)\}; (E \sqcap E_1 \sqcap E_2)^\omega; [\neg\mathsf{g}(E \sqcap E_1 \sqcap E_2)] \\ = & \{\text{Unfolding [7]}: \ S^\omega = S; S^\omega \sqcap \mathsf{skip}\} \\ & \{\mathsf{g}(E_1)\}; ((E \sqcap E_1 \sqcap E_2); (E \sqcap E_1 \sqcap E_2)^\omega \sqcap \mathsf{skip}); [\neg\mathsf{g}(E \sqcap E_1 \sqcap E_2)] \\ \sqsubseteq & \{\text{Refinement of } \sqcap\} \\ & \{\mathsf{g}(E_1)\}; E_1; (E \sqcap E_1 \sqcap E_2)^\omega; [\neg\mathsf{g}(E \sqcap E_1 \sqcap E_2)] \\ \sqsubseteq & \{\text{Refinement of } \sqcap \text{ and Lemma 9a in [7]}: \ S \sqsubseteq T \Rightarrow S^\omega \sqsubseteq T^\omega\} \\ & \{\mathsf{g}(E_1)\}; E_1; (E \sqcap E_2)^\omega; [\neg\mathsf{g}(E \sqcap E_1 \sqcap E_2)] \\ = & \{\text{Assumption 2 and rule}: \ \{g \cap h\} = \{g\}; \{h\}\} \\ & \{\mathsf{g}(E_1)\}; E_1; \{\mathsf{g}(E_2)\}; \{\neg\mathsf{g}(E_1)\}; (E \sqcap E_2)^\omega; [\neg\mathsf{g}(E \sqcap E_1 \sqcap E_2)] \\ = & \{\text{Assumption 1 and Lemma 14c in [7]}: \ S; T \sqsubseteq U; S \Rightarrow S; T^\omega \sqsubseteq U^\omega; S\} \\ & \{\mathsf{g}(E_1)\}; E_1; \{\mathsf{g}(E_2)\}; \{\neg\mathsf{g}(E_1)\}; (E \sqcap E_2)^\omega; \{\neg\mathsf{g}(E_1)\}; [\neg\mathsf{g}(E \sqcap E_1 \sqcap E_2)] \\ = & \{\text{Rules}: \ [g \cap h] = [g]; [h] \text{ and } \{g\}; [g] = \{g\}\} \\ & \{\mathsf{g}(E_1)\}; E_1; \{\mathsf{g}(E_2)\}; \{\neg\mathsf{g}(E_1)\}; (E \sqcap E_2)^\omega; \{\neg\mathsf{g}(E_1)\}; [\neg\mathsf{g}(E \sqcap E_2)] \\ \sqsubseteq & \{\text{Assertion removal}: \ \{g\} \sqsubseteq \mathsf{skip}\} \\ & \{\mathsf{g}(E_1)\}; E_1; \{\mathsf{g}(E_2)\}; (E \sqcap E_2)^\omega; [\neg\mathsf{g}(E \sqcap E_2)] \end{array}$$

That the refinement is non-miraculous can be shown using Lemma 17b in [7], $S^\omega; [\neg\mathsf{g}(S)](\mathsf{false}) = \mathsf{false}$, and $\{\mathsf{g}(S)\}; S(\mathsf{false}) = \mathsf{false}$. □

The pattern obtained from equation two from sched (45) is similar. The resulting proof obligation (47) can be proved in the same manner as (46) above. However, Assumption 2 need to be modified to take into account assertion $\{g\}$ instead of assertion $\{\mathsf{g}(E_2)\}$.

$$\{i\}; E_1 = \{i\}; E_1; \{i \cap \neg\mathsf{g}(E_1) \cap g\} \tag{54}$$

The last pattern obtained from equation five gives proof obligation (50). This pattern is used when the symbol hlt occurs after E_1 in the schedule. This proof obligation can be proved using unfolding [7], the rule $S^\omega \sqsubseteq \mathsf{skip}$ and the refinement rules for $\sqcap$.

Note that the verification conditions for the patterns are not local. That means the correctness of the patterns together with the assumptions about the events does not depend on only E_1 and E_2, but also on the rest of the events E. This is not the only way to prove these patterns correct, but the correctness will always in the end depend also on the events E.

This way of proving sequencing of events is not without problems. Consider again Assumption 1. We prove that each event does not become re-enabled. Assume we have n events that should be executed after each other. We need to show for each step that the event just scheduled is not re-enabled by any event that comes after. Furthermore, each condition of the form $g \subseteq (E_1 \sqcap E_2)(h)$ is divided into two separate proof obligations $g \subseteq E_1(h)$ and $g \subseteq E_2(h)$. Taking these properties into account, there will be $O(n^2)$ proof obligations (more exactly $n(n-1)/2$) from the schedule due to Assumption 1. For long sequences of events this means there are a huge number of proof obligations to prove.

Sequential composition in loops Here we like to show that we can sequence events inside a loop. Assume we have two sets of events E_1 and E_2. We are interested in scheduling them according to the scheduling pattern P_2.

$$\begin{array}{lll} P_2(E_1, E_2, S') \mathrel{\hat=} & & \\ \quad\text{Precondition} & : & \neg\mathsf{g}(E_2) \\ \quad\text{Schedule} & : & \{\neg\mathsf{g}(E_2)\} \to \mathbf{do}\ E_1 \to E_2 \to \mathsf{hlt}\ \mathbf{od}\ \to S' \\ \quad\text{Assumption 1} & : & \{i\}; E_1 = \{i\}; E_1; \{\mathsf{g}(E_2)\} \\ \quad\text{Assumption 2} & : & \{i\}; E_2 = \{i\}; E_2; \{\neg\mathsf{g}(E_2)\} \end{array} \tag{55}$$

Here we have the pre-condition $\neg\mathsf{g}(E_2)$ for the pattern, which again is used as a context assertion in the refinement proofs. The reason for the pre-condition will become clear when proving that the refinement is non-miraculous. Applying the function sched to the schedule we get:

$$\begin{array}{l} \{\neg\mathsf{g}(E_2)\}; \mathsf{sched}(E_1 \cup E_2 \cup E, S) \longrightarrow \\ \quad \{\neg\mathsf{g}(E_2)\}; ([\mathsf{g}(E_1 \sqcap E_2)]; \mathsf{sched}(E_1 \cup E_2, E_1 \to E_2 \to \mathsf{hlt}))^\omega; [\neg\mathsf{g}(E_1 \sqcap E_2)]; \\ \quad \mathsf{sched}(E, S') \end{array}$$

This step is verified by proof obligation (48). Here we focus on verification of the part of the pattern before S'. Applying the first instance of sched gives:

$$\mathsf{sched}(E_1 \cup E_2, E_1 \to E_2 \to \mathsf{hlt}) \longrightarrow E_1; \{\mathsf{g}(E_2)\}; E_2$$

After discharging proof obligation (48) and applying Lemma 4, we have to prove that:

$$\begin{array}{l} \{\neg\mathsf{g}(E_2)\}; (E_1 \sqcap E_2)^\omega; [\neg\mathsf{g}(E_1 \sqcap E_2)] \sqsubseteq \\ \{\neg\mathsf{g}(E_2)\}; ([\mathsf{g}(E_1 \sqcap E_2)]; E_1; \{\mathsf{g}(E_2)\}; E_2)^\omega; [\neg\mathsf{g}(E_1 \sqcap E_2)] \end{array}$$

This cannot be proved to be a refinement in general and Assumption 1 is therefore needed. This assumption states that E_1 must enable E_2. Using this assumption, the pattern can be proved correct.

Proof.

$$
\begin{array}{ll}
& \{\neg \mathsf{g}(E_2)\}; (E_1 \sqcap E_2)^\omega; [\neg \mathsf{g}(E_1 \sqcap E_2)] \\
= & \{\text{Decomposition (Lemma 12 in [7])}: \ (S \sqcap T)^\omega = S^\omega; (T; S^\omega)^\omega\} \\
& \{\neg \mathsf{g}(E_2)\}; E_2^\omega; (E_1; E_2^\omega)^\omega; [\neg \mathsf{g}(E_1 \sqcap E_2)] \\
\sqsubseteq & \{S^\omega \sqsubseteq \mathsf{skip}\} \\
& \{\neg \mathsf{g}(E_2)\}; (E_1; E_2^\omega)^\omega; [\neg \mathsf{g}(E_1 \sqcap E_2)] \\
= & \{\text{Assumption 1}\} \\
& \{\neg \mathsf{g}(E_2)\}; (E_1; \{\mathsf{g}(E_2)\}; E_2^\omega)^\omega; [\neg \mathsf{g}(E_1 \sqcap E_2)] \\
= & \{\text{Unfolding [7]}: \ S^\omega = S; S^\omega \sqcap \mathsf{skip}\} \\
& \{\neg \mathsf{g}(E_2)\}; (E_1; \{\mathsf{g}(E_2)\}; (E_2; E_2^\omega \sqcap \mathsf{skip}))^\omega; [\neg \mathsf{g}(E_1 \sqcap E_2)] \\
\sqsubseteq & \{\text{Refinement of } \sqcap \text{ and } S^\omega \sqsubseteq \mathsf{skip}\} \\
& \{\neg \mathsf{g}(E_2)\}; (E_1; \{\mathsf{g}(E_2)\}; E_2)^\omega; [\neg \mathsf{g}(E_1 \sqcap E_2)] \\
= & \{\mathsf{g}(E_1) \subseteq \mathsf{g}(E_1 \sqcap E_2) \text{ and } g \subseteq h \Rightarrow [g] = [h]; [g]\} \\
& \{\neg \mathsf{g}(E_2)\}; ([\mathsf{g}(E_1 \sqcap E_2)]; E_1; \{\mathsf{g}(E_2)\}; E_2)^\omega; [\neg \mathsf{g}(E_1 \sqcap E_2)]
\end{array}
$$

□

The obtained statement is not guaranteed to be non-miraculous, since the effective guard of the loop body is $\mathsf{g}(E_1)$. Assumption 2 which states that the event E_2 must disable itself, is needed to prove this property. The reason for this, perhaps unintuitive, assumption is that it guarantees that the guard of the loop body equals $\mathsf{g}(E_1 \sqcap E_2)$.

Proof.

$$
\begin{array}{ll}
& \{\neg \mathsf{g}(E_2)\}; ([\mathsf{g}(E_1 \sqcap E_2)]; E_1; \{\mathsf{g}(E_2)\}; E_2)^\omega; [\neg \mathsf{g}(E_1 \sqcap E_2)] \ (\mathsf{false}) \\
= & \{\text{Assumption 2}\} \\
& \{\neg \mathsf{g}(E_2)\}; ([\mathsf{g}(E_1 \sqcap E_2)]; E_1; \{\mathsf{g}(E_2)\}; E_2; \{\neg \mathsf{g}(E_2)\})^\omega; \\
& \quad [\neg \mathsf{g}(E_1 \sqcap E_2)] \ (\mathsf{false}) \\
= & \{\text{Leapfrog (Lemma 11 in [7])}: \ S; (T; S)^\omega = (S; T)^\omega; S\} \\
& (\{\neg \mathsf{g}(E_2)\}; ([\mathsf{g}(E_1 \sqcap E_2)]; E_1; \{\mathsf{g}(E_2)\}; E_2)^\omega; \{\neg \mathsf{g}(E_2)\}; \\
& \quad [\neg \mathsf{g}(E_1 \sqcap E_2)] \ (\mathsf{false}) \\
= & \{\text{Rule}: \ [g \cap h] = [g]; [h] \text{ and } \{g\} = \{g\}; [g]\} \\
& (\{\neg \mathsf{g}(E_2)\}; ([\mathsf{g}(E_1 \sqcap E_2)]; E_1; \{\mathsf{g}(E_2)\}; E_2)^\omega; \{\neg \mathsf{g}(E_2)\}; [\neg \mathsf{g}(E_1)] \ (\mathsf{false}) \\
\subseteq & \{\text{Assertion removal}: \ \{g\} \sqsubseteq \mathsf{skip}, \ \mathsf{g}(E_1) \subseteq \mathsf{g}(E_1 \sqcap E_2) \text{ and} \\
& \quad g \subseteq h \Rightarrow [g] = [h]; [g]\} \\
& ((E_1; \{\mathsf{g}(E_2)\}; E_2)^\omega; [\neg \mathsf{g}(E_1)] \ (\mathsf{false}) \\
= & \{\mathsf{g}(E_1; \{\mathsf{g}(E_2)\}; E_2) = \mathsf{g}(E_1) \text{ and rule (Lemma 17b in [7])}: \\
& \quad S^\omega; [\neg \mathsf{g}(S)](\mathsf{false}) = \mathsf{false}\} \\
& \mathsf{false}
\end{array}
$$

□

Note that here we proved the whole pattern in one step. We could also have proved the final sequential composition $E_2 \rightarrow \mathsf{hlt}$ using the pattern for introducing sequential composition.

Loops in sequential composition Consider again the pattern in (52), where we had a nested loop of events. Here we verify a variant of that scheduling pattern:

$$
\begin{array}{lll}
\multicolumn{3}{l}{P_3(E_1, E_2, S_1, S') \mathrel{\hat{=}}} \\
\quad\text{Precondition} & : & \mathsf{g}(E_1 \sqcap E_2) \\
\quad\text{Schedule} & : & \{\mathsf{g}(E_1 \sqcap E_2)\} \to \textbf{do } S_1 \textbf{ od } \to E_2 \to S' \\
\quad\text{Assumption 1} & : & \{i \cap \mathsf{g}(E_1 \sqcap E_2)\}; E_1 = \\
 & & \{i \cap \mathsf{g}(E_1 \sqcap E_2)\}; E_1; \{\mathsf{g}(E_1 \sqcap E_2)\} \\
\quad\text{Assumption 2} & : & \{\neg\mathsf{g}(E_1)\}; (E_2 \sqcap E) = \\
 & & \{\neg\mathsf{g}(E_1)\}; (E_2 \sqcap E); \{\neg\mathsf{g}(E_1)\}
\end{array}
\tag{56}
$$

This schedule differs from the one in (52) in that the outer loop should here be executed exactly once. The schedule can also be followed with several other events in S'. To make the schedule possible, we use pre-condition $\{\mathsf{g}(E_1 \sqcap E_2)\}$ for the pattern. This condition states that one of the events in the loop must be enabled. Application of function sched leads to the following statement:

$$
\begin{array}{l}
\{\mathsf{g}(E_1 \sqcap E_2)\}; \mathsf{sched}(E_1 \cup E_2 \cup E, S) \longrightarrow \\
\quad \{\mathsf{g}(E_1 \sqcap E_2)\}; ([\mathsf{g}(E_1)]; \mathsf{sched}(E_1, S_1))^\omega; [\neg\mathsf{g}(E_1)]; \{\mathsf{g}(E_2)\}; \\
\quad \mathsf{sched}(E_2 \cup E, E_2 \to S')
\end{array}
$$

This leads to the proof obligation (48), which we prove here for this pattern. We have the pre-condition $\{\mathsf{g}(E_1 \sqcap E_2)\}$ and we can therefore use Assumption 1 similarly to the corresponding assumption in pattern P_1 (52). We also need an assumption that events $E_2 \sqcap E$ do not enable E_1 again, similar to Assumption 1 needed in the sequential composition pattern.

Proof.

$$
\begin{array}{ll}
 & \{\mathsf{g}(E_1 \sqcap E_2)\}; (E_1 \sqcap E_2 \sqcap E)^\omega; [\neg\mathsf{g}(E_1 \sqcap E_2 \sqcap E)] \\
= & \{\text{Rule (Lemma 9c in [7])}: \ S^\omega = S^\omega; S^\omega\} \\
 & \{\mathsf{g}(E_1 \sqcap E_2)\}; (E_1 \sqcap E_2 \sqcap E)^\omega; (E_1 \sqcap E_2 \sqcap E)^\omega; [\neg\mathsf{g}(E_1 \sqcap E_2 \sqcap E)] \\
\sqsubseteq & \{\text{Refinement of } \sqcap \text{ and Lemma 9a in [7]}: \ S \sqsubseteq T \Rightarrow S^\omega \sqsubseteq T^\omega\} \\
 & \{\mathsf{g}(E_1 \sqcap E_2)\}; E_1^\omega; (E_2 \sqcap E)^\omega; [\neg\mathsf{g}(E_1 \sqcap E_2 \sqcap E)] \\
\sqsubseteq & \{\text{Assumption introduction}: \ \mathsf{skip} \sqsubseteq [g]\} \\
 & \{\mathsf{g}(E_1 \sqcap E_2)\}; E_1^\omega; [\neg\mathsf{g}(E_1)]; (E_2 \sqcap E)^\omega; [\neg\mathsf{g}(E_1 \sqcap E_2 \sqcap E)] \\
= & \{\text{Rule}: \ [g]; \{g\} = [g]\} \\
 & \{\mathsf{g}(E_1 \sqcap E_2)\}; E_1^\omega; [\neg\mathsf{g}(E_1)]; \{\neg\mathsf{g}(E_1)\}; (E_2 \sqcap E)^\omega; [\neg\mathsf{g}(E_1 \sqcap E_2 \sqcap E)] \\
\sqsubseteq & \{\text{Assumption 2 and Lemma 14c in [7]}: \ S; T \sqsubseteq U; S \Rightarrow S; T^\omega \sqsubseteq U^\omega; S\} \\
 & \{\mathsf{g}(E_1 \sqcap E_2)\}; E_1^\omega; [\neg\mathsf{g}(E_1)]; (E_2 \sqcap E)^\omega; \{\neg\mathsf{g}(E_1)\}; [\neg\mathsf{g}(E_1 \sqcap E_2 \sqcap E)] \\
= & \{\text{Rule}: \ \{g\} = \{g\}; [g] \text{ and } [g \cap h] = [g]; [h]\} \\
 & \{\mathsf{g}(E_1 \sqcap E_2)\}; E_1^\omega; [\neg\mathsf{g}(E_1)]; (E_2 \sqcap E)^\omega; \{\neg\mathsf{g}(E_1)\}; [\neg\mathsf{g}(E_2 \sqcap E)] \\
\sqsubseteq & \{\text{Assumption 1 and Lemma 14c in [7]}: \ S; T \sqsubseteq U; S \Rightarrow S; T^\omega \sqsubseteq U^\omega; S\} \\
 & \{\mathsf{g}(E_1 \sqcap E_2)\}; E_1^\omega; \{\mathsf{g}(E_1 \sqcap E_2)\}; [\neg\mathsf{g}(E_1)]; (E_2 \sqcap E)^\omega; \\
 & \quad \{\neg\mathsf{g}(E_1)\}; [\neg\mathsf{g}(E_2 \sqcap E)] \\
= & \{\text{Propagation of assertion and } \{g\}; \{h\} = \{g \cap h\} \\
 & \{\mathsf{g}(E_1 \sqcap E_2)\}; E_1^\omega; \{\mathsf{g}(E_1 \sqcap E_2)\}; [\neg\mathsf{g}(E_1)]; \{\neg\mathsf{g}(E_1) \cap \mathsf{g}(E_1 \sqcap E_2)\}; \\
 & \quad (E_2 \sqcap E)^\omega; \{\neg\mathsf{g}(E_1)\}; [\neg\mathsf{g}(E_2 \sqcap E)] \\
= & \{\text{Rule}: \ \mathsf{g}(S_1 \sqcap S_2) = \mathsf{g}(S_1) \cup \mathsf{g}(S_2) \text{ and set} - \text{theory}\} \\
 & \{\mathsf{g}(E_1 \sqcap E_2)\}; E_1^\omega; \{\mathsf{g}(E_1 \sqcap E_2)\}; [\neg\mathsf{g}(E_1)]; \{\neg\mathsf{g}(E_1) \cap \mathsf{g}(E_2)\}; (E_2 \sqcap E)^\omega; \\
 & \quad \{\neg\mathsf{g}(E_1)\}; [\neg\mathsf{g}(E_2 \sqcap E)] \\
\sqsubseteq & \{\text{Removal of assertions}: \ \{g\} \sqsubseteq \mathsf{skip}\} \\
 & \{\mathsf{g}(E_1 \sqcap E_2)\}; E_1^\omega; [\neg\mathsf{g}(E_1)]; \{\mathsf{g}(E_2)\}; (E_2 \sqcap E)^\omega; [\neg\mathsf{g}(E_2 \sqcap E)]
\end{array}
$$

That the refinement is non-miraculous can easily be shown using Lemma 17b in [7], $S^\omega; [\neg\mathsf{g}(S)](\mathsf{false}) = \mathsf{false}$. To finalize the verification of the pattern, we apply one of the sequential composition patterns e.g. P_{seq} in (53):

$$\{\mathsf{g}(E_2)\}; \mathsf{sched}(E_2 \cup E, E_2 \to S') = \{\mathsf{g}(E_2)\}; E_2; \mathsf{sched}(E, S')$$

$\square$

We have here given a collection of patterns that can be used to introduce complex control flow such as nested loops and sequential composition to Event-B specifications. The choice of patterns that are presented here is rather ad-hoc. There is no attempt to present the most useful patterns and the usefulness of the patterns has not been validated on larger case-studies. The focus has been on how patterns are developed and verified. These examples show how we can use assumptions about the events to prove that patterns are correct. Wherever in the schedule a schedule fragment matching the pattern occurs, we can introduce the statement obtained by applying the pattern. This requires that the assumptions the pattern rely on are fulfilled. To efficiently use the scheduling method, a library of scheduling patterns with associated verification conditions would be needed.

One might ask why the scheduling language and the patterns are needed at all. It would also be possible to directly use the algebraic rules to reason about the Event-B model as a whole. The problem with that approach is that the derivations and proofs have to be done for each developed program, which might be demanding especially for people who are not formal methods experts. The patterns expressed in the scheduling language encode reusable structures that have known verification conditions. The idea is that the scheduling method and patterns partition the scheduling problem into smaller parts. New patterns can then be applied separately on the parts themselves. This will hopefully make the scheduling problem more manageable. The hypothesis is that it is useful to develop a specification using an event based approach and then create a sequential program from that. However, assumptions used in the patterns need to be proved as well, which can sometimes be demanding.

4.4 After scheduling

The scheduling of events does not yet provide a deterministic program, only a statement that is known to be implementable. Here we give a number of patterns that can be applied to create an executable program from the statement obtained by scheduling the events. First recall that an event of the form $E =$ **when** G **then** S **end** corresponds to a set transformer $[E] = [g]; [s]$ as discussed in Section 2. Note that each set transformer $[s]$ needs to be deterministic in order for the whole program to be deterministic. The implementation patterns here are complete in the sense that all statements obtained by scheduling events can be implemented using them. The symbol $\rightsquigarrow$ denotes that the statement on the left hand side in the pattern can be replaced by the one on the right hand side.

If $[E] \mathrel{\hat{=}} [g]; [s]$ then we have the pattern:

$$\{\mathsf{g}(E)\}; [E] \rightsquigarrow [s] \tag{57}$$

since $\{\mathsf{g}([E])\}; [E] \sqsubseteq [s]$. In general if $[E] = [g_1]; [s_1] \sqcap \ldots \sqcap [g_n]; [s_n]$ then

$$\{\mathsf{g}([E])\}; [E] \rightsquigarrow \textbf{if } g_1 \textbf{ then } [s_1] \textbf{ elsif } g_2 \textbf{ then } [s_2] \ \ldots \textbf{else } [s_n] \textbf{ end} \tag{58}$$

since $\{\mathsf{g}([E])\}; [E] \sqsubseteq$ **if** g_1 **then** $[s_1]$ **elsif** g_2 **then** $[s_2]$ ...**else** $[s_n]$ **end** [6]. This is similar to the implementation pattern for choice between events given in [2]. Patterns can also be given for loops. The definition of while-loop **while** g **then** S **end** is given as $S^\omega; [\neg g]$ [6]. Due to how loops are verified in the schedule, they will always be of the form $S^\omega; [\neg \mathsf{g}([E_1])]$, where $\mathsf{g}([E_1])$ is the guard of some events E_1 in S. We then have the following pattern:

$$S^\omega; [\neg \mathsf{g}([E_1])] \rightsquigarrow \textbf{while } \mathsf{g}([E_1]) \textbf{ then } \{\mathsf{g}([E_1])\}; S \textbf{ end} \qquad (59)$$

Using these patterns sequential composition, non-deterministic choice, and iteration of events in the schedule can be implemented as imperative program flow constructs. Normal programming languages contain many more types of control flow, such as switch-case statements and for-loops. However, we have only introduced the most fundamental ones.

4.5 Example

Consider again the array summation example from Section 3. In the first case, when the summation is modelled by one event *progress*, the schedule is simple. The event is executed in a loop until it becomes disabled, which leads to the schedule **do** *progress* $\rightarrow$ hlt **od** $\rightarrow$ hlt. After applying Lemma 4 and the rule $[g] = [g]; \{g\}$, the scheduled statement is:

$$\begin{aligned}&\mathsf{sched}(\{progress\}, \textbf{do } progress \rightarrow \mathsf{hlt} \textbf{ od } \rightarrow \mathsf{hlt}) = \\ &\quad [progress]^\omega; [\neg \mathsf{g}([progress])]; \{\neg \mathsf{g}(progress)\};\end{aligned}$$

The scheduling of the initialisation and the original event *calc* is given by the structure of the action system that defines the Event-B model semantics (see Section 3). The result from applying the pattern for while-loop introduction is shown to the left below. Using the fact that we have already proved $\{\neg \mathsf{g}(progress)\} = \{\neg \mathsf{g}(progress)\}; \{\mathsf{g}(calc)\}$ (deadlock freeness proof obligation from Section 3) together with implementation patterns (57) and (59), the final program to the right is obtained.

```
[$initialisation$];
while $\mathsf{g}([progress])$
then
    $\{\neg\mathsf{g}(progress)\}; [progress]$
end ; $\{\neg\mathsf{g}(progress)\}$;
[$calc$]
```

```
$sum := 0$;
$s := 0$;
$j := 0$;
while $j < m$
then
    $j := j + 1$
    $s := s + f(j + 1)$
end ;
$sum := s$
```

In the second case where the summation is implemented as an unrolled loop we have the following schedule $progress_1 \rightarrow progress_2 \rightarrow progress_3 \rightarrow \mathsf{hlt}$. Using the pattern for sequential composition (53) three times the statement after scheduling becomes:

$$\begin{aligned}&\{\mathsf{g}([progress_1])\}; \mathsf{sched}(\{progress_1, progress_2, progess_3\}, \\ &\quad progress_1 \rightarrow progress_2 \rightarrow progress_3 \rightarrow \mathsf{hlt}) \\ &= \\ &\{\mathsf{g}([progress_1])\}; [progress_1]; \\ &\{\mathsf{g}([progress_2])\}; [progress_2]; \\ &\{\mathsf{g}([progress_3])\}; [progress_3]; \\ &\{\neg \mathsf{g}([progress_1]) \cap \neg \mathsf{g}([progress_2]) \cap \neg \mathsf{g}([progress_3])\};\end{aligned}$$

The assertion at the end comes from the fact that when $\mathsf{sched}(E, S)$ is viewed as not applied yet, then it has the interpretation $[E]^\omega; [\neg \mathsf{g}(E)]$. Together with the rule $[g] = [g]; \{g\}$ this means an assertion that states that none of the events are enabled can always be added at the end. The statement obtained from the Event-B model after scheduling is shown below to the left below. We have already proved that $i \cap \neg \mathsf{g}([\mathit{progress}_1]) \cap \neg \mathsf{g}([\mathit{progress}_2]) \cap \neg \mathsf{g}([\mathit{progress}_3]) \subseteq \mathsf{g}(\mathit{calc})$. This followed from the deadlock freeness proof obligation in Section 3 (assuming that the abstract version of *calc* was not guarded). Note that we still have the context assertion $\{\mathsf{g}([\mathit{progress}_1])\}$. In order to ensure that the statement refines the original Event-B model, we therefore have to prove the assumption $[\mathit{initialisation}] = [\mathit{initialisation}]; \{\mathsf{g}([\mathit{progress}_1])\}$. Using this property and implementation pattern (57) the following program to the right is obtained.

```
[$initialisation$] ;
{$\mathsf{g}([progress_1])$}; [$progress_1$];
{$\neg\mathsf{g}([progress_1])$};
{$\mathsf{g}([progress_2])$}; [$progress_2$];
{$\neg\mathsf{g}([progress_2])$};
{$\mathsf{g}([progress_3])$}; [$progress_3$];
{$\neg\mathsf{g}([progress_1]) \cap \neg\mathsf{g}([progress_2]) \cap$
  $\neg\mathsf{g}([progress_3])$}; [$calc$]
```

```
$sum := 0$;
$s := 0$;
$j := 0$;
$s := s + f(1); j := 1$;
$s := s + f(2); j := 2$;
$s := s + f(3); j := 3$
$sum := s$
```

Note that in this case there is an unnecessary scheduling variable j. This variable does not affect the computation of the value of *sum* and it would be desirable to remove it. This can be done as an extra data refinement step.

5 Data refinement

After the creation of the sequential program, data refinement can still be applied to the result. However, the proof tools for Event-B currently has no support for this type of development. Therefore, we like to use data refinement patterns that depend on pre-conditions that can be verified using syntactic checks. As seen above, it seems like there will often be scheduling variables that become unnecessary when the program has been scheduled. Data refinement can be used to remove these variables.

Assume we have a compound statement S assigning a variable x. If x is assigned a constant value and all sub-statements of S not assigning x are independent of x then we can move the assignment of x. We can move the final assignment of x such that $S = (x := c; S')$, where S' is equal to S but all assignments to x have been removed. The assignment $x := c$ assigns x the last value it was assigned in S. We can then use a data refinement

$$D; x := c; S' \sqsubseteq S''; D \tag{60}$$

Here the refinement statement used for this refinement is defined as $D(\phi) = \{((v \mapsto x) \mapsto v') | v = v' \land x = c\}[\phi]$. The statement S'' the same as S', but uses the variables v' instead of v, $S'[v'/v] = S''$.

In the summation example at the end of the previous section, the scheduling variable j is unnecessary. We can thus use the data refinement step above to remove j. The final program becomes:

```
$sum' := 0$;
$s' := 0$;
$s' := s' + f(1);\ s' := s' + f(2);\ s' := s' + f(3)$;
$sum' := s'$
```

The data refinement statement D used in this case is defined as $D(\phi) = \{(\mathit{sum} \mapsto s \mapsto j) \mapsto (\mathit{sum}' \mapsto s') | \mathit{sum}' = \mathit{sum} \land s' = s \land j = 3\}[\phi]$. We now have the desired program from Section 3.

6 Conclusions

This paper describes a method for deriving sequential programs from event based specifications. We first presented a suitable semantics for Event-B models to develop sequential programs, which was based on set transformers and action systems. A scheduling language was then presented for describing the flow of control. Throughout the paper, we used the algebraic approach from [7] to analyse the models and the scheduling. Using this approach we developed and verified different scheduling patterns. A simple example was also given to illustrate how the scheduling and the derivations are done in practise. The advantage of this approach is that much of the development of sequential programs can be carried out using the tools of Event B. Only the final scheduling step to introduce flow control constructs may require a few additional proof obligations.

Scheduling of events in Event-B or actions in action system has been done in CSP before [9, 10]. In this approach, program counters are introduced in the (Event-)B model to verify the scheduling. Here we use a more direct approach were we directly derive the needed proof obligations for a schedule. Our approach is not as flexible as scheduling using CSP, but it might be easier to apply since the condition under which the schedule is possible is explicit. It is also easy to derive proved reusable scheduling patterns using our approach, which might help the develop sequential code from the Event-B models in practice. Furthermore, the scheduled events map directly to imperative programming constructs. Scheduling of actions in action systems have also been done with a special scheduling language [14] for hardware synthesis. The language is similar to ours, but schedules actions within one iteration of the system. That means there are no loops in their schedules. They also use program counters to verify scheduling.

There are limitations to creating sequential programs from event based specifications in this way. The proof obligations needed by the scheduling can create a lot of extra work. For example, the pattern presented for sequential composition might problematic in practice due to the large number of proof obligations that would be generated. Another limitation is that there is currently no tool support to generate the proof obligations needed for the different patterns. The expressiveness of the scheduling language is also a limitation. For example, the choice operator $\|$ operates on events and not on statements. This was a design decision to simplify the methods in this paper. However, this limitation will be removed in the future. Furthermore, the scheduling language also has no support for structuring mechanisms such as procedures and modules.

This paper gives one approach how to schedule events and prove the correctness of schedules. It shows how scheduling patterns can be developed and proved. The algebraic approach used in the paper seems to be useful for reasoning about Event-B models on a higher level than the traditional proof obligations.

References

[1] J.-R. Abrial. Event driven sequential program construction. Clearsy, `http://www.atelierb.eu/php/documents-en.php`, 2001.

[2] J.-R. Abrial. Event based sequential program development: Application to constructing a pointer program. In *FME 2003: Formal Methods*, volume 2805 of *LNCS*, pages 51–74. Springer-Verlag, 2003.

[3] J. R. Abrial. *Modelling in Event B: System and Software Engineering*. Cambridge University Press, 2009. To appear.

[4] R.-J. R. Back and R. Kurki-Suonio. Decentralization of process nets with centralized control. In *Proceedings of the 2nd ACM SIGACT-SIGOPS Symposium of Principles of Distributed Computing*, pages 131–142, 1983.

[5] R.-J. R. Back and K. Sere. Stepwise refinement of action systems. *Structured Programming*, 12:17–30, 1991.

[6] R.-J. R. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer-Verlag, 1998.

[7] R.-J. R. Back and J. von Wright. Reasoning algebraically about loops. *Acta Informatica*, 36:295–334, 1999.

[8] R.-J. R. Back and J. von Wright. Encoding, decoding and data refinement. *Formal Aspects of Computing*, 12:313–349, 2000.

[9] M. Butler. Stepwise refinement of communicating systems. *Science of Computer Programming*, 27:139–173, 1996.

[10] M. Butler. csp2b: A practical approach to combining CSP and B. *Formal Aspects of Computing*, 12(3):182–198, 2000.

[11] F. Degerlund, M. Waldén, and K. Sere. Implementation issues concerning the action systems formalism. In *Eighth International Conference on Parallel and Distributed Computing, Applications and Technologies, 2007. PDCAT '07*, pages 471–479. IEEE, 2007.

[12] S. Hallerstede. On the purpose of Event-B proof obligations. In *Abstract State Machines, B and Z*, volume 5238 of *LNCS*, pages 125–138, 2008.

[13] Michael Leuschel and Michael Butler. ProB: An Automated Analysis Toolset for the B Method. *Journal Software Tools for Technology Transfer*, 10(2):185–203, 2008.

[14] J. Plosila, K. Sere, and M. Waldén. Asynchronous system synthesis. *Science of Computer Programming*, 55:259–288, 2005.

[15] Steve Wright. Automatic generation of C from Event-B. In *Workshop on Integration of Model-based Formal Methods and Tools*. `http://www.lina.sciences.univ-nantes.fr/apcb/IM_FMT2009/im_fmt2009_proceedings.html`, February 2009.

University of Turku
- Department of Information Technology
- Department of Mathematics

Åbo Akademi University
- Department of Computer Science
- Institute for Advanced Management Systems Research

Turku School of Economics and Business Administration
- Institute of Information Systems Sciences

ISBN 978-952-12-2338-9
ISSN 1239-1891