Concurrent Scheduling of Event-B Models *

Pontus Boström, Fredrik Degerlund, Kaisa Sere and Marina Waldén

Åbo Akademi University, Dept. of Information Technologies
Turku Centre for Computer Science (TUCS)
Joukahainengatan 3-5, FIN-20520 Åbo, Finland
{pontus.bostrom, fredrik.degerlund, kaisa.sere, marina.walden}@abo.fi

Event-B is a refinement-based formal method that has been shown to be useful in developing concurrent and distributed programs. Large models can be decomposed into sub-models that can be refined semi-independently and executed in parallel. In this paper, we show how to introduce explicit control flow for the concurrent sub-models in the form of event schedules. We explore how schedules can be designed so that their application results in a correctness-preserving refinement step. For practical application, two patterns for schedule introduction are provided, together with their associated proof obligations. We demonstrate our method by applying it on the dining philosophers problem.

1 Introduction

Event-B [1, 18] is a state-based modelling framework with its roots in the guarded command language and the Action Systems formalism [3, 4]. It advocates proof-based correct-by-construction design, abstraction, stepwise refinement and model decomposition as its main development strategies.

In an Event-B model, events are chosen non-deterministically for execution following the interleaving principle and assuming atomicity of events. Much of the effort in the refinement approach, especially down in the refinement chain, is about the modeller aiming at diminishing the non-determinism in the model and introducing more deterministic ways of choosing events for execution. In an extreme case we can think of the modeller encoding this by using explicit program counters in the events. Work on introducing more deterministic *schedules* of events to Event-B has been studied extensively recently [8, 11, 14, 20]. The goal has been to avoid explicitly coding this scheduling information into the events. We base our approach on [8], which concerns sequential systems, and extend it to concurrent programs.

When models become large, decomposition strategies are used to focus on specific parts of the model. To be practical, such strategies need to support compositional verification in the sense that the modeller can locally reason about properties of a decomposed part of the model even though the underlying Event-B assumption is that events are chosen for execution from the entire set of events in the model. Relying on the atomicity requirement for events and the interleaving semantics for Event-B models the distinct parts can be interpreted as concurrently executing models [12]. We show here how the scheduling approach of Boström [8] can be extended so that we can apply it in a compositional manner focusing only on part(s), or sub-model(s), of the model. We turn these sub-models into tasks, giving each of them a schedule of its own. The main addition to the original approach for sequential programs is to handle the possible interferences the concurrently executing tasks might exhibit. This can also be seen as an extension, with explicit schedules, of the Hoang-Abrial approach [12] to development of concurrent programs.

To facilitate practical use of our method, the schedules are introduced stepwise into a model via patterns. The patterns have associated proof obligations needed for ensuring the correctness of the refine-

---

ment step. As a result of the schedules, the scheduling information contained in events can be expressed explicitly in the schedules.

In this paper, we focus on developing concurrent programs following the stepwise refinement approach. Apart from the introduction of explicit schedules, concurrent programs are modelled within Event-B in a normal manner [1, 12]. While Event-B models can be executed as such using a non-deterministic scheduler ("animation"), our approach is designed to be close to traditional programming languages and results in models that are more efficient to execute on a computer, since more control flow information is explicitly stated in the schedule than using only Event-B [8]. The approach can also be used to replace parts of event behaviour with scheduling information as the scheduling concept as such is more general than what the focus is here. The schedules actually give a process-oriented specification style for Event-B modeller complementing its state-based style [9, 17].

The rest of this paper is structured as follows. In section 2, we present the foundations needed to understand our approach. We discuss set transformers (predicate transformers), the Event-B formalism and model decomposition. In section 3, we introduce a dining philosophers [13] Event-B model, which serves as a running example. Section 4 presents our main contributions. We introduce a scheduling language, show how schedules and tasks can be introduced, and demonstrate how it is possible to tackle the problem of interference from interleaving tasks. In section 5, we show how our framework can be applied on the dining philosophers example model. Finally, we sum the paper up in section 6, where we also discuss related work and future perspectives.

2 Foundations

2.1 Event-B

Event-B [1, 10] is a state-based modelling language. Models in Event-B consist of a dynamic and a static part, referred to as *machines* and *contexts*, respectively. The most important parts of a machine are *variables*, an *invariant* and *events*. Contexts contain parts such as *constants*, which can be referred to from machines. The state space is made up of the variables $v_1, ..., v_n$ of types $\Sigma_1, ..., \Sigma_n$, and can be modelled as the cartesian product $\Sigma = \Sigma_1 \times ... \times \Sigma_n$. The events $E_1, ..., E_m$ modify the state space, and can be written in the following general form [10], where $k \in 1..m$:

$$E_k \mathrel{\widehat{=}} \textbf{when } G_k(v,c) \textbf{ then } v : |A_k(v,v',c) \textbf{ end}. \tag{1}$$

Here, v represents the variables, c the constants seen by the machine, and the *action* $v : |A_k(v,v',c)$ is the nondeterministic assignment assigning v any such values v' for which $A_k(v,v',c)$ holds. $G_k(v,c)$ represents the *guard*, which is a condition that must hold in order for the action to take place. An event is said to be *enabled* when its guard holds. Each machine also contains a special event *Initialisation* $\mathrel{\widehat{=}} v : |A_0(v',c)$ that initialises the state space. Unlike other events, it is unguarded and does not depend on a previous state. Events can be classified as *ordinary*, *convergent* or *anticipated*. This will be further explained in section 2.4. The invariant $I(v,c)$ is a predicate constraining the values of the variables.

2.2 Set transformers

The events in Event-B can be viewed as set transformers [10]. Our presentation of events as set transformers is similar to the presentation in [10].

Consider a state space Σ. A set transformer is a function $\mathscr{P}(\Sigma) \to \mathscr{P}(\Sigma)$ that tranforms a set of states into another set of states. A weakest precondition set transformer S applied to a set q returns the largest set p from which S is guaranteed to reach a state in q.

We have the following definitions to give a set transformer semantics to Event-B models:

$$
\begin{aligned}
\Sigma &= \{v\,|\,\top\} \\
i &= \{v\,|\,I(v,c)\} \\
g_k &= \{v\,|\,G_k(v,c)\} \\
a_k &= \{v \mapsto v'\,|\,A_k(v,v',c)\} \\
a_0 &= \{v'\,|\,A_0(v',c)\}
\end{aligned}
\tag{2}
$$

The set i describes the subset of the state space where the invariant I holds. Similarly, the sets g_k ($k \in 1..m$) represent the state space subsets where guard G_k of the respective event E_k is true. The relation a_k describes the possible before-after states that can be achieved by the assignment of the respective event. Note that the initialisation results in a set a_0 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 at this level of reasoning. The axioms that describe the properties of the constants are here considered to be part of the invariant.

Let g and q be subsets of Σ, and a be a relation. Furthermore, S, S_1 and S_2 are arbitrary set transformers. The variables of Σ are denoted v. We have the following set transformers:

$$[a](q) \mathrel{\hat=} \{v\,|\,a[\{v\}] \subseteq q\} \qquad \text{(Nondeterministic update)} \tag{3}$$

$$[g](q) \mathrel{\hat=} \neg g \cup q \qquad \text{(Assumption)} \tag{4}$$

$$\{g\}(q) \mathrel{\hat=} g \cap q \qquad \text{(Assertion)} \tag{5}$$

$$(S_1 \sqcap S_2)(q) \mathrel{\hat=} S_1(q) \cap S_2(q) \qquad \text{(Nondeterministic choice)} \tag{6}$$

$$S_1; S_2(q) \mathrel{\hat=} S_1(S_2(q)) \qquad \text{(Sequential composition)} \tag{7}$$

$$S^\omega(q) \mathrel{\hat=} \mu X.(S; X \sqcap \mathsf{skip})(q) \qquad \text{(Strong iteration)} \tag{8}$$

$$S^*(q) \mathrel{\hat=} \nu X.(S; X \sqcap \mathsf{skip})(q) \qquad \text{(Weak iteration)} \tag{9}$$

$$\mathsf{skip}(q) \mathrel{\hat=} q \qquad \text{(Stuttering)} \tag{10}$$

$$\mathsf{magic}(q) \mathrel{\hat=} \mathsf{true} \qquad \text{(Miracle)} \tag{11}$$

$$\mathsf{abort}(q) \mathrel{\hat=} \mathsf{false} \qquad \text{(Aborting)} \tag{12}$$

Here, true and false are notations representing the sets Σ and $\emptyset$, respectively. This is because of convenience as well as the fact that the same notation is used in weakest precondition predicate transformers. We will also in general use predicate notation for describing subsets of the state space. (Nondeterministic) update is used to assign values to variables in the state space, of which the stuttering set transformer skip is a special case, which leaves the state unmodified. The set transformer magic achieves the desired postcondition (even false) from any state, whereas abort does not guarantee to achieve any postcondition q from any state. Not even termination is guaranteed. Assumption and assertion both behave as skip when g is true, but when false, assumption behaves as magic, whereas assertion behaves as abort. Nondeterministic choice represents demonic choice between set transformers, and sequential composition combines set transformers in a sequential manner. An important property of demonic choice is that miraculous behaviour is avoided whenever possible, whereas aborting behaviour is always preferred. This is demonstrated by the following theorems, which follow directly from the definitions:

$$
\begin{aligned}
\mathsf{magic} \sqcap S &= S \\
\mathsf{abort} \sqcap S &= \mathsf{abort}
\end{aligned}
\tag{13}
$$

The following properties can easily be derived, and the proofs can also be found in [5]:

$$
\begin{aligned}
&\text{magic}; S = \text{magic} && \text{abort}; S = \text{abort} \\
&\{g\}; [h] = \{g\} && [g]; \{h\} = [g] && (14)\\
&\{g \cap h\} = \{g\}; \{h\} && [g \cap h] = [g]; [h]
\end{aligned}
$$

The iteration set transformers are used to achieve repeated execution. Iteration has been thoroughly discussed by Back and von Wright [5, 6], and is only shortly summarised here. In both strong and weak iteration (S^ω and S^*, respectively), the set transformer S is repeatedly executed a demonically chosen number of times. In strong iteration, the number of executions may be infinite, whereas for weak iteration it is guaranteed to be finite. Important theorems regarding iteration include the following *unfolding* rules:

$$
\begin{aligned}
S^\omega &= S; S^\omega \sqcap \text{skip} \\
S^* &= S; S^* \sqcap \text{skip}
\end{aligned} \tag{15}
$$

The set of states in which a set transformer S does not behave miraculously is called the guard of S. The guard $g(S)$ is given as:

$$
g(S) \mathrel{\hat{=}} \neg S(\text{false}) \tag{16}
$$

We can now interpret an event E_k from (1) as a set transformer. Using the definitions from (2), we can now give the set transformer $[E_k]$ for E_k as [10]:

$$
[E_k] \mathrel{\hat{=}} [g_k]; [a_k] \tag{17}
$$

For a set of events, $\{E_1, \ldots, E_m\}$, we will use the denotion $[E]$ for the expression $[E_1] \sqcap \ldots \sqcap [E_m]$.

2.3 Refinement

Refinement is an important concept in Event-B. In this paper, we are mainly interested in refinement on the set transformer level, where it can be defined as [5]:

$$
S_1 \sqsubseteq S_2 \mathrel{\hat{=}} \forall s. S_1(s) \subseteq S_2(s) \tag{18}
$$

Here, S_1 and S_2 are set transformers. The intuitive interpretation of $S_1 \sqsubseteq S_2$ is that if S_1 will reach a state in a set s, then so will S_2. We say that S_1 and S_2 are (refinement) equivalent if and only if $S_1 \sqsubseteq S_2$ and $S_2 \sqsubseteq S_1$. The relation between the set transformer view of refinement and a proof obligations approach has been studied in [10].

A set transformed S is said to behave miraculously when executed in a state in the set $S(\text{false})$, i.e. when the execution of S results in a post-state belonging to the empty set. We typically want to avoid introduction of more miraculous behaviour during refinement. Given a set transformer S_1 and a refinement S_2, S_2 does not exhibit more miraculous behaviour than S_1 if $S_1(\text{false}) = S_2(\text{false})$.

2.4 Behavioural semantics

We aim at using Event-B for construction of concurrent programs. Ultimately we like to show that a (concurrent) program S is correct given a precondition P and a postcondition Q. This correctness requirement is expressed in the Hoare triple:

$$
\{P\} \, S \, \{Q\} \tag{19}
$$

As the basis for our method, we use the development method for concurrent programs in [12]. In this approach, the concurrent programs are built from atomic events in the same way as sequential programs are constructed [1]. The program *S* is considered to consist of a collection of events. Note that there is no control flow other than non-deterministic choice of enabled events. Using the refinement based approach of Event-B, the program *S* that satisfies the pre/post-specification is derived stepwise. In order to use the refinement process to develop programs, the pre-/post-specification first has to be encoded into an initial Event-B model. This model has a specific structure [1]: it has an initialisation event *init*, progress events *prog* and a finalisation event *fin*. The events *prog* model (non-deterministically) the computation of the program, while *fin* models the post-condition *Q* as a guard. The precondition is encoded in an external context machine. The semantics of an Event-B model *M* specifying a sequential program is in this setting:

$$M \triangleq [init] ; [prog]^* ; [fin] \tag{20}$$

The system is first initialised, then *prog* is executed until the postcondition given by *fin* becomes true. The program can then terminate. The progress events *prog* are later refined to create a deterministic algorithm to reach the postcondition. We will also later need to show that the refinements *E* of *prog* terminate [1], i.e. $[E]^{\omega} = [E]^*$, as we are interested in total correctness. We assume that all Event-B models in the rest of the paper have this structure. Each event should maintain the invariant and therefore we assume that there is an invariant assertion $\{i\}$ implicitly given before and after each event.

We previously mentioned that events can be classified as *ordinary*, *convergent* or *anticipated*. This is relevant from a behavioural semantics point of view. Events are normally classified as ordinary, but it is sometimes necessary to prove that execution of events from a group will eventually terminate. All events belonging to this group should then be labelled as convergent. In practice, the termination property is proven by introducing a variant, and by showing that it is decreased by all convergent events. There is also the possibility of classifying events as anticipated. Labelling an event as anticipated indicates that it will be classified as convergent in a later refinement step, whereby the proof is postponed until further down the refinement chain. The notions anticipated or convergent should be for the events *prog* to guarantee that the model eventually terminates.

2.5   Decomposition

In order for a refinement based development method to be scalable there should be a way to decompose specifications into smaller parts that can be independently developed. The verification of refinement should thus be compositional, i.e., refinement of the individual parts should yield a refinement of the whole system.

Here we will use a decomposition approach based on shared variables [1, 2]. Following this approach, a model can be decomposed into sub-models that can themselves be further decomposed. The set of sub-models forms the complete system model.

Definition 1. *Sub-model. A sub-model is given as a 7-tuple* (*v*,*x*,*E*,*X*,*I*,*init*,*fin*)*, where v and x are sets of variables, E and X are sets of events, I the invariant, init the initialisation and fin the finalisation.*

The variables *v* are only visible inside the sub-model, and will be referred to as internal variables. Variables *x* are shared with other components and will be called external variables. The events *E* can refer to both *v* and *x*. Since they (also) manipulate the internal variables of the sub-model, they are denoted the internal events. The external events, *X*, are abstractions that only refer to the external variables *x* modelling the effects of events of other components. Hence, each event in *X* has a corresponding internal event in another component. The initialisation of a sub-model is given by event *init* and the loop

termination guard is given by event *fin*. Note that a traditional Event-B model can be seen as a sub-model where the sets of external events and external variables are empty. A sub-model $(v,x,E,X,I,init,fin)$ can be (further) decomposed into sub-models:

$$(v,x,E,X,I,init,fin) = (v_1,x_1,E_1,X_1,I_1,init_1,fin_1) \parallel (v_2,x_2,E_2,X_2,I_2,init_2,fin_2)$$

The parallel composition of the sub-models is defined as:

$$
\begin{aligned}
& (v_1,x_1,E_1,X_1,I_1,init_1,fin_1) \parallel (v_2,x_2,E_2,X_2,I_2,init_2,fin_2) \\
\hat{=}\ & (v_1 \cup v_2, (x_1 \cup x_2) \backslash (v_1 \cup v_2), E_1 \cup E_2, (X_1 \cup X_2) \backslash (E_1 \cup E_2), I_1 \wedge I_2, init_1 \parallel init_2, fin_1 \parallel fin_2)
\end{aligned}
\tag{21}
$$

The parallel composition of two events is given as:

$$
\begin{aligned}
& \textbf{when } G \textbf{ then } v : |S \textbf{ end} \parallel \textbf{when } H \textbf{ then } w : |R \textbf{ end} \\
\hat{=}\ & \textbf{when } G \wedge H \textbf{ then } v,w : |S \wedge R \textbf{ end}
\end{aligned}
\tag{22}
$$

The semantics $[M_1 \parallel M_2]$ of a the parallel composition $M_1 \parallel M_2$ is given as:

$$[M_1 \parallel M_2] \ \hat{=}\ init_1 \parallel init_2; ([E_1 \cup E_2 \cup ((X_1 \cup X_2) \backslash (E_1 \cup E_2))])^*; [\neg g(fin_1 \parallel fin_2)] \tag{23}$$

The composition can be extended to arbitrary many components by recursively merging components pairwise. Since we want to do compositional proofs of refinement, we need to show that refinement of the individual sub-models lead to refinement of the entire system. First we need to prove that the external events provide abstractions of their internal counterparts $\{i_1 \cap i_2\}; [X_1] \sqsubseteq [E_2] \sqcap [X_2]$ and $\{i_1 \cap i_2\}; [X_2] \sqsubseteq [E_1] \sqcap [X_1]$. To compositionally prove the refinement $[M_1 \parallel M_2] \sqsubseteq [M_1' \parallel M_2]$, we then only need to prove the refinement $[M_1] \sqsubseteq [M_1']$, see [7].

We need to model that external events are executed a finite number of times, as they model the finite execution of their internal counterparts in other sub-models. Since these external events are not necessarily terminating by themselves, strong iteration cannot be used for describing behaviour of sub-models. The use of weak iteration can be seen as compositionally verifying partial correctness of a program, since termination is not ensured by set transformer refinement. However, we want to prove total correctness of the complete system. Since we in this approach [1, 12] label the events E as anticipated or convergent, we show that the model will eventually terminate. Hence, total correctness follows from partial correctness in combination with the Event-B proof obligations that ensure termination [5, 6].

3   Dining philosophers case study

3.1   Problem description

We are now ready to introduce a model of the dining philosophers [13], which will serve as a running example. In this section, we show the initial model, we refine it, as well as decompose it into sub-models. The dining philosophers scenario can be described as follows. There are four philosophers sitting around a round table. Each philosopher has a plate in front of him, and there is a fork placed between each pair of adjacent plates. Each philosopher always does one of two things: think and eat, but not both at the same time. Furthermore, in order to eat, a philosopher must pick up both of the two forks located next to his plate. A philosopher can also drop a fork back into its original position, but only after he has eaten.

The basic problem is that if the philosophers pick up the forks arbitrarily, there may be deadlocks. For example, if each philosopher picks up his right fork, there will not be any forks available anymore,

and no philosopher will have enough forks to eat. Since a philosopher will not drop a fork until he has eaten, there will be a deadlock. One well-known solution to this problem is to assign a number to each fork, and enforce that each philosopher picks up the adjacent fork with the lowest number first. In our case study we assume that we have four philosophers and number the forks as follows: Philosopher 1 can access forks 1 and 2, philosopher 2 accesses forks 2 and 3, philosopher 3 uses forks 3 and 4, while philosopher 4 has access to forks 1 and 4.

3.2   Modelling and refinement

Initially we model the scenario as an abstract Event-B machine, where the four philosophers eat in a non-deterministic order. We only model one round, so each philosopher will only eat once. We introduce the variables *ph1eaten* thru *ph4eaten*, to model whether each philosopher has eaten. The event *Intialisation* sets these variables to FALSE. The events *Ph1Eat* thru *Ph4Eat* for the four philosophers then represent the progress of the model. They model that a philosopher eats which has not yet eaten by setting the corresponding variable to TRUE. Finally, event *Finalisation* checks that all four philosophers have eaten. The *Initialisation* and *Finalisation* events are classified as ordinary events, whereas *Ph1Eat*, ..., *Ph4Eat* are convergent, since they correspond to the *prog* variables in (20). We now have:

variables	**invariant**	Initialisation (*ordinary*) $\hat{=}$
ph1eaten	*ph1eaten* $\in$ BOOL	**begin**
ph2eaten	*ph2eaten* $\in$ BOOL	*ph1eaten* := FALSE
ph3eaten	*ph3eaten* $\in$ BOOL	*ph2eaten* := FALSE
ph4eaten	*ph4eaten* $\in$ BOOL	*ph3eaten* := FALSE
		ph4eaten := FALSE
		end

Ph1Eat (*convergent*) $\hat{=}$
when
   ph1eaten = FALSE
then
   ph1eaten := TRUE
end

Finalisation (*ordinary*) $\hat{=}$
when
   ph1eaten = TRUE
   ph2eaten = TRUE
   ph3eaten = TRUE
   ph4eaten = TRUE
then
   skip
end

In the first refinement step we introduce the forks, which are modelled as variables *fork1* thru *fork4*. They are of type 0..4 to represent which philosopher that currently holds the fork. Value 0 represents the fork lying on the table. All forks are initialised to this value. There are 16 new events in this refinement step: two for each of the four philosophers getting their adjacent forks (e.g. *Ph3GetFork3* and *Ph3GetFork4*), and two events for each philosopher releasing the corresponding forks (e.g. *Ph3RelFork4* and *Ph3RelFork3*). Note that philosopher 4 uses forks 1 and 4.

In order to be able to prove that the new events will not take over the execution, we classify them as convergent and give a variant that they decrease. There is no variable that can be used as a variant, but when each new event is executed it will disable itself and it will not be enabled again. Hence, we define a function *v* as follows:

$$v = \{ \quad (FALSE, FALSE, FALSE) \mapsto 5,$$
$$(TRUE, FALSE, FALSE) \mapsto 4,$$
$$(TRUE, TRUE, FALSE) \mapsto 3,$$
$$(TRUE, TRUE, TRUE) \mapsto 2,$$
$$(TRUE, FALSE, TRUE) \mapsto 1,$$
$$(FALSE, FALSE, TRUE) \mapsto 0\}$$

The first and second dimension of the triple correspond to whether a philosopher is holding his left or right fork, respectively. The third one indicates whether he has already eaten or not. The variant is then formed as a sum of the values of function v applied on the variables of each philosopher. The refined model is now as follows:

variables
  fork1
  fork2
  fork3
  fork4
  ph1eaten
  ph2eaten
  ph3eaten
  ph4eaten

invariant
  $fork1 \in 0..4$
  $fork2 \in 0..4$
  $fork3 \in 0..4$
  $fork4 \in 0..4$
  …
variant
  $v(bool(fork1=1), bool(fork2=1), ph1eaten)$
    $+v(bool(fork2=2), bool(fork3=2), ph2eaten)$
    $+v(bool(fork3=3), bool(fork4=3), ph3eaten)$
    $+v(bool(fork1=4), bool(fork4=4), ph4eaten)$

Initialisation (*ordinary*) $\;\hat{=}$
begin
  $fork1 := 0$
  $fork2 := 0$
  $fork3 := 0$
  $fork4 := 0$
  $ph1eaten := FALSE$
  $ph2eaten := FALSE$
  $ph3eaten := FALSE$
  $ph4eaten := FALSE$
end

Ph1GetFork1 (*convergent*) $\;\hat{=}$
when
  $fork1 = 0$
  $ph1eaten = FALSE$
then
  $fork1 := 1$
end

Ph1GetFork2 (*convergent*) $\;\hat{=}$
when
  $fork1 = 1$
  $fork2 = 0$
  $ph1eaten = FALSE$
then
  $fork2 := 1$
end

Ph1Eat (*convergent*) $\;\hat{=}$
when
  $fork1 = 1$
  $fork2 = 1$
  $ph1eaten = FALSE$
then
  $ph1eaten := TRUE$
end

Ph1RelFork2 (*convergent*) $\;\hat{=}$
when
  $fork2 = 1$
  $ph1eaten = TRUE$
then
  $fork2 := 0$
end

Ph1RelFork1 (*convergent*) $\;\hat{=}$
when
  $fork2 = 0$
  $fork1 = 1$
  $ph1eaten = TRUE$
then
  $fork1 := 0$
end

Finalisation (*ordinary*) $\;\hat{=}$
when
  $fork1 = 0$
  $fork2 = 0$
  $fork3 = 0$
  $fork4 = 0$
  $ph1eaten = TRUE$
  $ph2eaten = TRUE$
  $ph3eaten = TRUE$
  $ph4eaten = TRUE$
then
  skip
end

Note that when the v function is called, the fork variables are not directly passed as parameters. Instead, we check whether the currently evaluated philosopher holds the fork or not. The *bool* function is a technicality of Event-B that is needed to convert the result of the comparison into a value of BOOL.

The events corresponding to philosophers 2, 3 and 4 eating, as well as picking up and releasing their respective forks are analogous to the events of philosopher 1, and are thus not shown here. We now have a refined model for the four philosophers eating, and in the next subsection we will decompose this model.

3.3   Decomposition

In the decomposition step we separate the functionality of the four philosophers in such a way that each philosopher constitutes a sub-model of its own. The partitioning we achieve is shown in the table below. Since philosophers 2 and 4 share fork 2 and fork 1, respectively, with philosopher 1, the external events of sub-model 1 are Ph2GetFork2, Ph2RelFork2, Ph4GetFork1 and Ph4RelFork1. Analogous reasoning is used to find the external events of the other sub-models.

	Sub-model 1	Sub-model 2	Sub-model 3	Sub-model 4
Internal events	Ph1Eat	Ph2Eat	Ph3Eat	Ph4Eat
	Ph1GetFork1	Ph2GetFork2	Ph3GetFork3	Ph4GetFork1
	Ph1RelFork1	Ph2RelFork2	Ph3RelFork3	Ph4RelFork1
	Ph1GetFork2	Ph2GetFork3	Ph3GetFork4	Ph4GetFork4
	Ph1RelFork2	Ph2RelFork3	Ph3RelFork4	Ph4RelFork4
External events	Ph2GetFork2	Ph1GetFork2	Ph2GetFork3	Ph1GetFork1
	Ph2RelFork2	Ph1RelFork2	Ph2RelFork3	Ph1RelFork1
	Ph4GetFork1	Ph3GetFork3	Ph4GetFork4	Ph3GetFork4
	Ph4RelFork1	Ph3RelFork3	Ph4RelFork4	Ph3RelFork4

4   Concurrent programs

This far, we have considered model decomposition, resulting in sub-models that can be refined semi-independently. We are now ready to examine how these sub-models can be executed in a concurrent or parallel setting. This problem has been studied in [12], which is a case study showing how to decompose Event-B models into concurrently executing sub-models. Here we extend this approach by giving sub-models explicit flow control in the form of event schedules, instead of the traditional nondeterministic choice. An important concept in our approach is the concept of *tasks*, which we define as follows:

Definition 2. *Task. A task is an 8-tuple* $(v, x, E, X, I, init, fin, S)$ *where v are the internal variables, x the external variables, E the internal events, X the external events, I the invariant, init the initialisation, fin the loop termination condition, and S is a schedule conforming to the syntax in (24) concerning the internal events E.*

Since all coordinates, except for S, are the same as in a sub-model, a task can be seen as an extension of the sub-model concept. Whereas the events of traditional decomposed sub-models are executed non-deterministically, the internal events of a task are scheduled according to S. The schedule S may only consist of internal events, and the set of events in the schedule is denoted $e(S)$. We assume that $E = e(S)$, since if an internal event was not included in the schedule, it would never be executed.

4.1   Scheduling language

In order to describe schedules of events we give a small scheduling language [8], which adheres to the following syntax:

$$
\begin{aligned}
S &\;::=\; PS \to S \mid PS \\
PS &\;::=\; \textbf{do } S \textbf{ od} \mid S_1 \,[\!] \ldots [\!]\, S_n \mid E \mid \{g\}
\end{aligned}
\tag{24}
$$

Here $\to$ represents sequential composition, $[\!]$ non-deterministic choice, **do od** is a loop, E an event and $\{g\}$ is an assertion.

4.2   Semantics of tasks

The semantics of schedules is given using a function sched that maps each schedule to the corresponding set transformer as in [8]. However, when scheduling the events in a task we need to consider interference from other tasks. A goal of the scheduling language is to be able to express schedules of internal events in such a way that interference from external events does not have to be explicitly taken into account. Such interference freedom is instead proven separately. We now recursively define a function $\mathsf{sched}(S,X)$ where S is a schedule, X is the set of external events.

$$
\begin{aligned}
\mathsf{sched}(PS \to S, X) &= \mathsf{sched}(PS, X); \mathsf{sched}(S, X) \\
\mathsf{sched}(\textbf{do } S \textbf{ od}, X) &= ([\mathsf{g}([\mathsf{e}(S) \cup X])]; \mathsf{sched}(S, X))^*; [\neg \mathsf{g}([\mathsf{e}(S) \cup X])] \\
\mathsf{sched}(S_1 \,[\!] \ldots [\!]\, S_n, X) &= \mathsf{sched}(S_1, X) \sqcap \ldots \sqcap \mathsf{sched}(S_n, X) \\
\mathsf{sched}(E, X) &= [X]^*; [E]; [X]^* \\
\mathsf{sched}(\{g\}, X) &= \{g\}
\end{aligned}
\tag{25}
$$

The scheduling function takes the schedule S, as well as the set of external events X as input and outputs a set transformer containing both internal and external events. An arbitrary (but finite) number of external events X can occur before and after an internal event E in a schedule. This is modelled by the set transformer $[X]^*$ on both sides of the event.

Consider a system consisting of two tasks $T_1 = (v_1, x_1, E_1, X_1, init_1, fin_1, S_1)$ and $T_2 = (v_2, x_2, E_2, X_2, init_2, fin_2, S_2)$. To find the complete system behaviour, we need to compose the tasks, i.e. obtain $T_1 \parallel T_2$. However, the number of interleavings of atomic set transformers grows exponentially with the length of the schedule [19]. Hence, we need an appropriate approach to reason about the interleavings in order to make refinement proofs manageable. Here we make the restriction that we only consider tasks where the set transformers obtained after scheduling can be decomposed into a loop containing the demonic choice of atomic set transformers. This is an extension of the approach used in [12], where the programs are built from atomic *events* that are chosen non-deterministically for execution. Composition of such tasks can be easily handled [7]. We have the following requirement for schedulability in our approach:

$$
\exists S_{11}, \ldots, S_{1n} \cdot \mathsf{sched}(S_1, X_1) = (S_{11} \sqcap \ldots \sqcap S_{1n} \sqcap [X_1])^*; [fin_1]
\tag{26}
$$

where all S_{1i} are atomic compositions of internal events. Using these atomic set transformers we can now use the traditional parallel composition [7]. The semantics of the composition of the whole system $T_1 \parallel T_2$ is now given as:

$$
[T_1 \parallel T_2] \;\widehat{=}\; [init_1 \parallel init_2]; ((\sqcap_i S_{1i}) \sqcap (\sqcap_j S_{2j}))^*; [fin_1 \parallel fin_2]
\tag{27}
$$

This approach thus extends the decomposition method in [2, 12] with the possibility to reason about groups of sequentially scheduled events, instead of only individual ones. However, to find the groups

$S_{11}, \ldots, S_{1n}$ is in general non-trivial. Here we will give special cases encoded as *patterns* to make the verification of schedules manageable in practise.

4.3   Introduction of schedules

Schedules are introduced for the sub-models as a refinement step, in which we convert sub-models into tasks. The introduction of schedules has to constitute a refinement step in order to ensure that the properties we have already proved for the models before introduction of schedules are preserved. Note that we do not support scheduling of anticipated events, so they have to be turned into convergent ones before the introduction of schedules.

We now need to show for the two tasks $T_1 = (v_1, x_1, E_1, X_1, init_1, fin_1, S_1)$ and $T_2 = (v_2, x_2, E_2, X_2, init_1, fin_1, S_2)$:

$$[M_1 \parallel M_2] \sqsubseteq [T_1 \parallel T_2] \tag{28}$$

where sub-model M_i corresponds to task T_i as $M_i = (v_i, x_i, E_i, X_i, init_i, fin_i)$. As in the traditional decomposition method, we can use external events to perform compositional proofs of refinement. Here we rely on the property (26) to decompose schedule $\mathsf{sched}(S_i, X_i)$ into a loop consisting of atomic set transformers. We need to show that for all tasks T_i [7]:

$$\{i_1 \cap i_2\}; [X_{ij}] \sqsubseteq S_{kj} \tag{29}$$

$$([\mathsf{e}(S_i)] \sqcap [X_i])^*; [fin_i] \sqsubseteq \mathsf{sched}(S_i, X_i) \tag{30}$$

In (29) we assume that for any external event $X_{ij} \in X_i$, there is one corresponding atomic set transformer S_{kj} in another task T_k. To give a practical approach to the decomposition of schedules required by (26), we give patterns that give generic instantiations of the quantified variables. In the patterns we rely on special cases of scheduling constructs where we know we can prove (29) and (30). Patterns thus encode reusable schedule structures. One such case is when the introduction of sequential behaviour does not alter the behaviour of the sub-model. Another useful special case is when the introduction of sequential behaviour does not modify the externally visible behaviour of a sub-model. We use the same scheduling approach as in [8], where patterns are applied on schedules stepwise and we prove that each pattern application leads to a refinement of the previous application.

A pattern consists of a *precondition*, a *schedule*, a *result* and a number of *assumption*. The precondition predicate describes under which conditions the pattern is applicable. The schedule part describes what schedule the pattern is intended for, and the result part gives the set transformer that is produced when the pattern is applied. The assumptions are extra conditions that have be fulfilled in order to use the pattern.

Pattern 1   The first pattern, P_1, introduces sequential behaviour into a sub-model.

$$
\begin{aligned}
&P_1(E_1, h, g, S, X) \; \hat{=} \\
&\quad \text{Precondition} \quad : \; h \\
&\quad \text{Schedule} \qquad : \; E_1 \to \{g\} \to S \\
&\quad \text{Result} \qquad\;\; : \; \{h\}; X^*; E_1; X^*; \{g\}; \mathsf{sched}(S, X) \\
&\quad \text{Assumption 1} \; : \; h \subseteq \neg g(\mathsf{e}(S)) \\
&\quad \text{Assumption 2} \; : \; g \subseteq \neg g(E_1) \\
&\quad \text{Assumption 3} \; : \; \{g\}; (X \sqcap \mathsf{e}(S)) \sqsubseteq (X \sqcap \mathsf{e}(S)); \{g\} \\
&\quad \text{Assumption 4} \; : \; \{h\}; X \sqsubseteq X; \{h\} \\
&\quad \text{Assumption 5} \; : \; E_1 = E_1; \{g\}
\end{aligned}
\tag{31}
$$

The first assumption states that the precondition h implies that the events following E_1 are disabled. The second assumption states that g ensures that E_1 is disabled. Context information cannot be propagated in schedules without taking interference into account. Hence we need assumptions 3 and 4 to state that g and h are invariant with respect to the environment. Furthermore, g should also be invariant for all events in the schedule S. The last assumption states that E_1 will establish g. We also directly use the event name E_1 instead of the set transformer $[E_1]$, as well as E instead of $[E]$.

In order to stepwise use patterns we need to show that each application of a pattern is correct, i.e. that (30) holds. In order to do that, we assume that $\mathsf{sched}(S,X)$ represents a yet unscheduled loop of events $\mathsf{sched}(S,X) = (\mathsf{e}(S) \sqcap X)^*; [\mathsf{g}(\mathsf{e}(S) \sqcap X)]$. We instantiate the existential quantifier in (26) with S_i as E_i. Hence, we then need to show that $\{h\}; \mathsf{sched}(E_1 \rightarrow \{g\} \rightarrow S) = \{h\}; X^*; E_1; X^*; \{g\}; \mathsf{sched}(S)$. Note that we also rely here on the properties (32)-(34) in Lemma 1. Note also that to ensure (30) we here assume $i \cap \neg\mathsf{g}(E \sqcap X) \subseteq \mathsf{g}(fin)$. The reason for formulating the pattern in this way is to be able to use the same verification approach also to nested loops.

Lemma 1. *Context preservation. If $\{g\}; S \sqsubseteq S; \{g\}$ then:*

$$\{g\}; S = \{g\}; S; \{g\} \tag{32}$$

$$\{g\}; S^* = \{g\}; S^*; \{g\} \tag{33}$$

$$\{g\}; S^* = (\{g\}; S)^* \tag{34}$$

The proofs of the properties in the lemma are straightforward and they are omitted for brevity. We can now prove the correctness of pattern P_1.

Proof.

$$
\begin{aligned}
& \{h\}; \mathsf{sched}(E_1 \rightarrow \{g\} \rightarrow S, X); [\neg\mathsf{g}(E_1 \sqcap E \sqcap X)] \\
= \ & \{\text{Representation of } \mathsf{sched}(E_1 \rightarrow \{g\} \rightarrow S)\} \\
& \{h\}; (E_1 \sqcap E \sqcap X)^*; [\neg\mathsf{g}(E_1 \sqcap E \sqcap X)] \\
= \ & \{\text{Decomposition } [6] : (S \sqcap T)^* = (S; T^*)^*; T^*\} \\
& \{h\}; X^*; (E_1 \sqcap E; X^*)^*; [\neg\mathsf{g}(E_1 \sqcap E \sqcap X)] \\
= \ & \{\text{Distributivity}\} \\
& \{h\}; X^*; ((E_1; X^*) \sqcap (E; X^*))^*; [\neg\mathsf{g}(E_1 \sqcap E \sqcap X)] \\
= \ & \{\text{Decomposition}\} \\
& \{h\}; X^*; ((E_1; X^*)^*; ((E; X^*); (E_1; X^*)^*)^*; [\neg\mathsf{g}(E_1 \sqcap E \sqcap X)] \\
= \ & \{\text{Unfolding (15)}\} \\
& \{h\}; X^*; ((E_1; X^*); (E_1; X^*)^*) \sqcap \mathsf{skip}; ((E; X^*); (E_1; X^*)^*)^*; [\neg\mathsf{g}(E_1 \sqcap E \sqcap X)] \\
= \ & \{\text{Assumption 3 and Property (33)}\}\} \\
& \{h\}; X^*; \{h\}; (E_1; X^*); (E_1; X^*)^*) \sqcap \{h\}; ((E; X^*); (E_1; X^*)^*)^*; [\neg\mathsf{g}(E_1 \sqcap E \sqcap X)] \\
= \ & \{\text{Distributivity, assumption } h \subseteq \neg\mathsf{g}(E) \text{ and disabledness of guard}\} \\
& \{h\}; X^*; \{h\}; (E_1; X^*); (E_1; X^*)^*; ((E; X^*); (E_1; X^*)^*)^*; [\neg\mathsf{g}(E_1 \sqcap E \sqcap X)] \\
= \ & \{\text{Assumption } E_1 = E_1; \{g\}\} \\
& \{h\}; X^*; \{h\}; E_1; X^*; \{g\}; (E_1; X^*)^*; ((E; X^*); (E_1; X^*)^*)^*; [\neg\mathsf{g}(E_1 \sqcap E \sqcap X)] \\
= \ & \{\text{Assumption } g \subseteq \neg\mathsf{g}(E_1)\} \\
& \{h\}; X^*; \{h\}; E_1; X^*; \{g\}; (E; X^*; (E_1; X^*)^*)^*; [\neg\mathsf{g}(E_1 \sqcap E \sqcap X)]
\end{aligned}
$$

$$= \quad \{\text{Property (34) and } * \text{ below}\}$$
$$\{h\};X^*;\{h\};E_1;\{g\};X^*;\{g\};(\{g\};E;X^*;\{g\})^*;[\neg g(E_1 \sqcap E \sqcap X)]$$
$$= \quad \{\text{Leapfrog } [6] : S;(T;S)^* = (S;T)^*;S\}$$
$$\{h\};X^*;\{h\};E_1;\{g\};X^*;(\{g\};E;X^*)^*;\{g\};[\neg g(E_1 \sqcap E \sqcap X)]$$
$$= \quad \{\text{Assumption } g \subseteq \neg g(E_1) \text{ and } \{g\};[g] = \{g\}\}$$
$$\{h\};X^*;\{h\};E_1;\{g\};X^*;(\{g\};(E;X^*))^*;\{g\};[\neg g(E \sqcap X)]$$
$$= \quad \{\text{Lemma 9(c) in } [6] : S^* = S^*;S^* \text{ and decomposition}\}$$
$$\{h\};X^*;\{h\};E_1;\{g\};X^*;(\{g\};E \sqcap \{g\};X)^*;[\neg g(E \sqcap X)]$$
$$= \quad \{\text{Property (33) and assumption 5}\}$$
$$\{h\};X^*;E_1;X^*;\{g\};(\{g\};E \sqcap \{g\};X)^*;[\neg g(E \sqcap X)]$$
$$= \quad \{\text{Representation of } \text{sched}(S,X)\}$$
$$\{h\};X^*;E_1;X^*;\{g\};\text{sched}(S,X)$$

The proof of step $*$ is:

$$(\{g\};E;X^*;(E_1;X^*)^*)^*$$
$$= \quad \{\text{Assumption 3 and Properties (32) and(33)}\}$$
$$(\{g\};E;X^*;\{g\};(E_1;X^*)^*)^*$$
$$= \quad \{\text{Assumption 2}\}$$
$$(\{g\};E;X^*;\{g\})^*$$

$$\square$$

Pattern 2   The second pattern, P_2, also introduces sequential behaviour. However, this time we show that we can group local behaviour E_2 to an arbitrary event.

$$
\begin{aligned}
&P_2(E_1,E_2,h,g,S_1,X) \triangleq \\
&\text{Precondition} \quad : \quad h \\
&\text{Schedule} \qquad : \quad E_1 \to E_2 \to \{g\} \to S \\
&\text{Result} \qquad\quad : \quad \{h\};X^*;E_1;X^*;E_2;X^*;\{g\};\text{sched}(S,X) \\
&\text{Assumption 1} \ : \quad h \subseteq \neg g(e(S)) \\
&\text{Assumption 2} \ : \quad g \subseteq \neg g(E_1 \sqcap E_2) \\
&\text{Assumption 3} \ : \quad E_2;X = X;E_2 \\
&\text{Assumption 4} \ : \quad \{g(E_2)\};X = X;\{g(E_2)\} \\
&\text{Assumption 5} \ : \quad \{g\};(X \sqcap e(S)) \sqsubseteq (X \sqcap e(S));\{g\} \\
&\text{Assumption 6} \ : \quad \{h\};X \sqsubseteq X;\{h\} \\
&\text{Assumption 7} \ : \quad E_2 = E_2;\{g\}
\end{aligned}
\tag{35}
$$

The assumptions in pattern P_2 are similar to the ones in P_1. However, we additionally need assumptions that states that E_2 and X do not interfere with each other (assumptions 3 and 4). To prove the correctness of the pattern we need to show that

- By instantiation of (26) we get: $\{h\};X^*;E_1;X^*;E_2;X^*;\{g\};\text{sched}(S,X) = \{h\};(E_1;E_2 \sqcap e(S) \sqcap X)^*;[\neg g(E_1 \sqcap E_2 \sqcap e(S) \sqcap X)]$

- Refinement (30): $\{h\};\text{sched}(E_1 \to E_2 \to \{g\} \to S,X) \sqsubseteq \{h\};(E_1;E_2 \sqcap e(S) \sqcap X)^*;[\neg g(E_1 \sqcap E_2 \sqcap e(S) \sqcap X)]$

- Deadlock freedom: $\{h\};(E_1;E_2 \sqcap e(S) \sqcap X)^*;[\neg g(E_1 \sqcap E_2 \sqcap e(S) \sqcap X)](\text{false}) = \{h\};\text{sched}(E_1 \to E_2 \to \{g\} \to S,X)(\text{false})$

The deadlock freedom proof obligation ensures that the scheduling does not introduce new deadlocks. This was not needed in pattern P_1, as that pattern does not alter the behaviour of models. The proofs are straightforward using the assumptions in the pattern. This ensures that the scheduling does not introduce more deadlocks than in the original system.

5 Scheduling of dining philosophers

We now return to the running example introduced in section 3. Up till now, the dining philosophers model has been refined and split into sub-models. Now, we show how the sub-models can be turned into tasks by introducing schedules. In the scheduling process we use the patterns given in section 4.3. Correctness will be proven by checking the assumptions of the patterns. We will concentrate on how to derive a schedule for task 1. The schedules for task 2, 3 and 4 can be derived analogously.

Our approach is that the schedule should be formulated such that it fulfills the previously mentioned solution to the dining philosophers problem, i.e., that each philosopher should pick up the lower numbered fork first. Since we first want to pick up fork number 1, we wish to schedule *Ph1GetFork1* as the first event. The correct order of events will be *Ph1GetFork1*, *Ph1GetFork2*, *Ph1Eat*, *Ph1RelFork2*, *Ph1RelFork1*. This is captured by the following schedule:

$$\text{Ph1GetFork1} \rightarrow \{g_1\} \rightarrow \text{Ph1GetFork2} \rightarrow \text{Ph1Eat} \rightarrow \{g_2\}$$
$$\rightarrow \text{Ph1RelFork2} \rightarrow \{g_3\} \rightarrow \text{Ph1RelFork1} \rightarrow \{g_4\}$$

The assertions in the schedule are needed to capture intermediate results and thereby enable verification of the schedule in smaller parts.

We now want to prove that it is correct to schedule *Ph1GetFork1* as the first event. To show this, we will follow pattern P_1 introduced in Section 4.3 and show that the assumptions 1 - 5 for the pattern are fulfilled. We instantiate pattern P_1 as $P_1(Ph1GetFork1, h_1, g_1, S_r, X_{t1})$, where $h_1 = (fork1 \neq 1 \wedge ph1eaten = FALSE)$, $g_1 = (fork1 = 1 \vee ph1eaten = TRUE)$, $S_r = \text{Ph1GetFork2} \rightarrow \text{Ph1Eat} \rightarrow \{g_2\} \rightarrow \text{Ph1RelFork2} \rightarrow \{g_3\} \rightarrow \text{Ph1RelFork1} \rightarrow \{g_4\}$ and $X_{t1} = \{\text{Ph2GetFork2, Ph4GetFork1, Ph2RelFork2, Ph4RelFork1}\}$.

We chose precondition h_1 so that it also is an invariant for the external events X_{t1}. Here, h_1 states that philosopher 1 does not hold his forks nor has he eaten. Moreover, we chose assertion g_1 to state that philosopher 1 has picked up fork 1 or eaten. This condition is an invariant for the events $\mathsf{e}(S_r) \cup X_{t1}$ and established by *Ph1GetFork1*. We now confirm that the assumptions for the pattern hold:

- $h_1 = (fork1 \neq 1 \wedge ph1eaten = FALSE)$ implies that events in $\mathsf{e}(S_r)$ are disabled. This holds, since they are only enabled when philosopher 1 holds fork 1 or has eaten.

- The assertion $g_1 = (fork1 = 1 \vee ph1eaten = TRUE)$ following event *Ph1GetFork1* ensures that *Ph1GetFork1* is disabled. Since g_1 is a negation of the guard of *Ph1GetFork1* the second assumption is fulfilled.

- g_1 is an invariant of the environment $\mathsf{e}(S_r) \cup X_{t1}$. This is fulfilled, since in the events of $\mathsf{e}(S_r)$ philosopher 1 holds fork 1 or has eaten. Moreover, the events in X_{t1} that share fork 1 are not enabled when philosopher 1 holds fork 1, and none of these events modify variable *ph1eaten*.

- h_1 is an invariant of the external events X_{t1}. Since none of the external events model that philosopher 1 picks up fork 1 or modify variable *ph1eaten*, this assumption holds.

- Event *Ph1GetFork1* establishes g_1. This holds trivially since *Ph1GetFork1* models that philosopher 1 picks up fork 1 ($fork1 := 1$).

To verify the complete schedule, we then apply pattern P_2 once, followed by three applications of P_1. In the last application of P_1, the schedule following the assertion is empty. This can be interpreted as a schedule with an event that is always disabled. When task 1 has been fully proven, the whole procedure is repeated to schedule tasks 2, 3 and 4 in the order shown in the table below (for simplicity, the assertions are not shown).

Task 1	Task 2	Task 3	Task 4
Ph1GetFork1	Ph2GetFork2	Ph3GetFork3	Ph4GetFork1
$\rightarrow$ Ph1GetFork2	$\rightarrow$ Ph2GetFork3	$\rightarrow$ Ph3GetFork4	$\rightarrow$ Ph4GetFork4
$\rightarrow$ Ph1Eat	$\rightarrow$ Ph2Eat	$\rightarrow$ Ph3Eat	$\rightarrow$ Ph4Eat
$\rightarrow$ Ph1RelFork2	$\rightarrow$ Ph2RelFork3	$\rightarrow$ Ph3RelFork4	$\rightarrow$ Ph4RelFork4
$\rightarrow$ Ph1RelFork1	$\rightarrow$ Ph2RelFork2	$\rightarrow$ Ph3RelFork3	$\rightarrow$ Ph4RelFork1

6   Conclusions and related work

In this paper, we have proposed a method of correct-by construction development of concurrent programs using Event-B. The programs are first developed as proposed by Hoang and Abrial [12]. From this development process we obtain a number of sub-models that communicate via shared variables, which represent the program. We then introduce explicit control flow in the form of schedules for each sub-model, so that each sub-model/schedule corresponds to exactly one task. The schedules are introduced as correctness preserving refinements. We use a set-transformer semantics for Event-B, as well as well known algebraic rules [6] for the analysis of correctness. The schedules are verified in a step-wise manner, and each step carries some related proof obligations. The schedules enable more efficient implementation of the Event-B models as more explicit control flow information is available than for pure event-B models. We can, e.g., use the transformations in [8] to introduce traditional control flow constructs, such as while loops and if-statements, as well as remove unnecessary guards. Furthermore, the schedules give a process-oriented specification of the behaviour of the models.

Our goal is to compositionally reason about concurrent programs. This has been a very active field of research [19]. Our approach directly extends the approach in [12] for development of concurrent programs with explicit schedules of events. Compositional reasoning in this setting goes back to the work of Owicki and Gries [16] and Jones' Rely-Guarantee reasoning [15]. The decomposition method based on shared variables in Event-B [2, 12] is based on these ideas. Essentially the same approach is also available for action systems using the refinement calculus [7]. The theory for decomposition in the set-transformer setting is largely based on that paper. Several approaches to introducing control flow into Event-B models have been developed. Hallerstede's approach in [11] to adding control flow only deals with sequential programs and it is thus more related to Boström's earlier work [8]. The scheduling approaches in [14, 20] can also handle concurrent schedules. In [14] the scheduling (referred to as *flows*) is expressed using a special purpose language, while in the approach [20] the scheduling is expressed in CSP. The latter approach can be seen as an extension of the former. Processes or flows are both considered to communicate via shared events. Our focus is on compositional verification and scheduling of concurrent programs that use shared variables for communication. However, in both approaches not all events need to be scheduled, but non-scheduled events are considered interleaved in the scheduled. This could be used to take into account external events, and thus be used for compositional verification of shared variable programs also. Our contribution is threefold: 1) Compared to purely event-based modelling, we consider explicit schedules of events that can be interleaved 2) We do all analysis on the level of set transformers, which gives convenient formalism to algebraically perform the needed analysis

of Event-B models 3) We provide patterns and a method to develop patterns for introducing control flow in a stepwise manner. This is important, since verifying that a certain event schedule is correct can be very challenging and reusable scheduling structures can significantly aid in this task.

Set-transformers give a powerful framework to reason about Event-B models on a high level of abstraction. They give a good basis for creating reusable patterns for scheduling, which are essential for practical applications. If schedules are introduced as a last refinement step, as in the example of this paper, existing tool support can be used for development up till, but not including, the scheduling step. Future work involves investigating tool support for schedule application. Generation of refinement proof obligations for scheduled models is also of interest, since that would allow for schedule intoduction earlier in the refinement chain.

References

[1] J. R. Abrial (2010): *Modeling in Event B: System and Software Engineering*. Cambridge University Press.

[2] J.-R. Abrial & S. Hallerstede (2007): *Refinement, Decomposition, and Instantiation of Discrete Models: Application to Event-B*. Fundamenta Informaticae 77(1-2), pp. 1–28.

[3] R.-J. R. Back & R. Kurki-Suonio (1983): *Decentralization of Process Nets with Centralized Control*. In: *Proceedings of the 2nd ACM SIGACT-SIGOPS Symposium of Principles of Distributed Computing*, pp. 131–142, doi:10.1145/800221.806716.

[4] R.-J. R. Back & K. Sere (1991): *Stepwise Refinement of Action Systems*. Structured Programming 12, pp. 17–30.

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

[6] R.-J. R. Back & J. von Wright (1999): *Reasoning algebraically about loops*. Acta Informatica 36, pp. 295–334, doi:10.1007/s002360050163.

[7] R.-J. R. Back & J. von Wright (2003): *Compositional Action System Refinement*. Formal Aspects of Computing 15, pp. 103–117, doi:10.1007/s00165-003-0005-6.

[8] P. Boström (2010): *Creating sequential programs from Event-B models*. In: *IFM'10 Proceedings of the 8th international conference on Integrated formal methods*, LNCS 6396, Springer-Verlag, pp. 74–88, doi:10.1007/978-3-642-16265-7_7.

[9] M. Butler (2000): *csp2B: A Practical Approach to Combining CSP and B*. Formal Aspects of Computing 12(3), pp. 182–198, doi:10.1007/PL00003930.

[10] S. Hallerstede (2008): *On the purpose of Event-B proof obligations*. In: *Abstract State Machines, B and Z*, LNCS 5238, Springer-Verlag, pp. 125–138, doi:10.1007/978-3-540-87603-8_11.

[11] S. Hallerstede (2010): *Structured Event-B models and proofs*. In: *Abstract State Machines, B and Z*, LNCS 5977, Springer-Verlag, pp. 273–286, doi:10.1007/978-3-642-11811-1_21.

[12] T. S. Hoang & J-.R. Abrial (2010): *Event-B decomposition for parallel programs*. In: *ABZ2010*, LNCS 5977, Springer-Verlag, pp. 319–333, doi:10.1007/978-3-642-11811-1_24.

[13] C.A.R. Hoare (1985): *Communicating Sequential Processes*. Prentice Hall.

[14] A. Iliasov (2009): *On Event-B and control flow*. Technical Report CS-TR-1159, School of Computing Science, Newcastle University.

[15] C. B. Jones (1983): *Tentative steps toward a development method for interfering programs*. Transactions on Programming languages and systems 5(4), pp. 596–619, doi:10.1145/69575.69577.

[16] S. S. Owicki & D. Gries (1976): *An axiomatic proof technique for parallel programs I*. Acta Informatica 6, pp. 319–340, doi:10.1007/BF00268134.

[17] J. Plosila, K. Sere & M. Waldén (2005): *Asynchronous system synthesis*. Science of Computer Programming 55, pp. 259–288, doi:10.1016/j.scico.2004.05.018.

[18] (2010): *Rodin Platform*. `http://www.event-b.org`.

[19] W. P. de Roever & et. al. (2001): *Concurrency Verification: Introduction to compositional and noncompositional methods*. Cambridge University Press.

[20] S. Schneider, H. Treharne & H. Wehrheim (2010): *A CSP Approach to Control in Event-B*. In: *Integrated Formal Methods 2010*, *LNCS* 6396, Springer-Verlag, pp. 260–274, doi:10.1007/978-3-642-16265-7_19.