Theoretical Computer Science ••• (••••) •••–•••

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

Schedulers and finishers: On generating and filtering the behaviours of an event structure ☆

Annabelle McIver [a], Tahiry Rabehaja [a,*], Georg Struth [b]

[a] Department of Computing, Macquarie University, Australia
[b] Department of Computer Science, The University of Sheffield, United Kingdom

ARTICLE INFO

Article history:
Received 13 April 2017
Received in revised form 12 January 2018
Accepted 15 January 2018
Available online xxxx

Keywords:
Concurrency
Event structure
Partial order
Completeness
Specification

ABSTRACT

It is well known that every trace of a transition system can be generated using a scheduler. However, this basic completeness result does not hold in event structure models. The reason for this failure is that, according to its standard definition, a scheduler chooses which action to schedule and, at the same time, observes that the one scheduled last has occurred. Thus, scheduled events will never be able to overlap. We propose to separate scheduling from observing termination and introduce the dual notion of finishers which, together with schedulers, are enough to regain completeness. We then investigate all possible interactions between schedulers and finishers, concluding that simple alternating interactions are enough to express complex resolution. We also observe that when these interactions are independent, they may produce behaviours that are not satisfying some desired property that is intrinsic to the system. To filter these behaviours out, we extend our results by defining permissible pairs of schedulers and finishers. In contrast to independent interactions, this new concept allows us to control and observe concurrent executions with a granularity that is strictly higher than that provided by the bundle relation.

1. Introduction

Formal software analysis is principally based on the meaning given to computations. Often this semantics is defined as the set of behaviours a program can perform. For instance, the sequential behaviours of a labelled transition system are given by traces. In general, these behaviours are generated by schedulers. For labelled transition systems, schedulers are complete in the sense that each and every trace of the system can be generated by a scheduler. This completeness, however, fails if we are to model truly-concurrent behaviours using event structures. This paper introduces the concept of *finishers* in order to complement schedulers and thus provide a complete technique for the generation of all the behaviours of an arbitrary event structure.

A trace belonging to the language of a labelled transition system systematically records a totally ordered sequence of actions that are performed sequentially over time. In other words, a new action, to be appended at the end of the trace, cannot start unless the last action in that trace has terminated. The total order between the actions captures exactly one of the behaviours of a sequential (or interleaved) system. However, there are cases where we need to model situations with

☆ This research was supported by the ARC Discovery Grant DP1092464 and the EPSRC Grant EP/J003727/1.
* Corresponding author.
E-mail address: tahiry.rabehaja@mq.edu.au (T. Rabehaja).

https://doi.org/10.1016/j.tcs.2018.01.015
0304-3975/

overlapping actions [20], or parallel executions with inter-process communication [13]. Totally ordering the actions fails to capture these situations faithfully and the most natural solution is to weaken the total ordering of actions into a partial ordering of events [4,13,15,24,27,30]. Thus, the behaviours of an event structure are encoded as labelled partially ordered sets, or *lposets* for short, where comparable events must occur in the given order and incomparable ones are concurrent. These concurrent events may happen in any order (interleaving) or they may overlap (true-concurrency).

Every trace of a labelled transition system can be generated by a scheduler. Intuitively, the scheduler walks through the transition system and resolves all choices by selecting one of the next available actions based on the execution history. The same technique can be defined for event structures but it is not complete because such a scheduler forces sequential dependencies, specified by the order in which events are scheduled. In other words, the scheduler does two different jobs in one go: it determines which events have occurred and which are scheduled to happen. By assigning the first task to a different entity, which we call *finishers*, we are able to schedule an event without observing the termination of the actions associated to previously scheduled events.

In this paper, the sole role of a scheduler is to choose an event that is available or *enabled*, given the current history of the computation encoded as a lposet. Once an event is scheduled, its associated action is considered to be ready to run or has started to execute but *not* yet terminated. Observing and remembering termination is the job of a finisher. Intuitively, a finisher looks at a lposet corresponding to the scheduled events ordered with causal dependencies, and observes which part of that lposet has safely occurred. Thus, a finisher has at least two basic properties: finished events must have been scheduled sometime in the past and they must remain finished as the computation progresses. Formally, these two properties correspond to contraction and monotonicity of a finisher.[1]

Through this dichotomy, we show that each and every behaviour of an event structure results from the interaction between a scheduler and finisher, which implies the completeness of the behaviour generation method.

In this extension of our previous work [18], we also explore the expressiveness of the interaction between schedulers and finishers. We study a more restricted form of interaction to control the amount of concurrent events in the generated behaviours. Concretely, we work with two particular examples; the first one is based on mutual exclusion using a simple locking mechanism. In this example, we assume that two sequential processes are concurrently trying to access their critical sections. These critical sections are bounded by locking (this happens when the lock is successfully acquired) and unlocking events (this happens when the lock is successfully released). The two processes are mutually exclusive when the bounded sections never overlap. In practice, the non-overlapping requirement is ensured if we have a sequential dependency between the last unlocking event and the scheduled locking event. Such a behaviour can only be obtained with specific pairs of schedulers and finishers because that dependency is not encoded within the order relation from the underlying event structure. That is, independent interactions between schedulers and finishers, which are crucial for the completeness theorem (Theorem 1), will generate all possible behaviours including those that do not satisfy the mutual exclusion property. To filter out these unsafe behaviours, we only resolve the event structure with respect to the above specific pairs of schedulers and finishers, which we refer to as *permissible* pairs. Surprisingly, a permissible pair of scheduler and finisher is also required to provide explicit sequential dependencies between all locking events, in addition to the aforementioned dependency between locking and the most recent unlocking events.

Our second example encodes inter-process communication using schedulers and finishers. Process communication are achieved via channels through which processes are sending and receiving messages. In our example, we consider a very weak form of concurrency in the style of Misra [20]. That is, different processes are allowed to send different messages through the same channel. Our only constraint is that "a message being read must have been sent some time in the past" [14]. We show how finishers are able to formally capture such an informal intuition. Moreover, for this particular example, we show that the new dependencies generated from the *permissible* scheduler and finisher interaction can be pushed back into the order relation from the event structure.

The main contributions of this extended paper are listed below.

1. We give an insight into the basic nature of schedulers and introduce finishers to account for the dual counterpart of scheduling (Sec. 3.2).
2. We show that schedulers and finishers provide a complete technique to generate each and every behaviour of an event structure (Theorem 1). This technique gives a novel operational perspective at the dynamics of event structures.
3. We show that all complex interactions between schedulers and finishers can be obtained from simple alternating interactions (Theorem 2).
4. We show how to use *permissible* schedulers and finishers to filter behaviours satisfying intrinsic characteristics including mutual exclusion and message passing (Sec. 6).

This paper is organised as follows. Sec. 2 gives a summary of the important notions related to event structures. Sec. 3 introduces schedulers and finishers whose alternating interactions are elaborated in Sec. 4 to generate all the behaviours of an event structure. Sec. 5 shows that arbitrary interactions between schedulers and finishers can be expressed using the simpler alternating interactions. Sec. 6 exposes how we use schedulers and finishers to express intrinsic properties of true-

[1] Given a poset $(X, \leq)$ and a function $f{:}X \to X$, f is a contraction (resp. monotonic) if $f(x) \leq x$ (resp. $x \leq y$ implies $f(x) \leq f(y)$).

Please cite this article in press as: A. McIver et al., Schedulers and finishers: On generating and filtering the behaviours of an event structure, Theoret. Comput. Sci. (2018), https://doi.org/10.1016/j.tcs.2018.01.015