Software Modelling for Partied Scheduling and Real-time Protocols

Bohua Zhan
State Key Lab. of Computer Science, Institute of Software
Chinese Academy of Sciences, Beijing, China

Yi Lv
State Key Lab. of Computer Science, Institute of Software
Chinese Academy of Sciences, Beijing, China

Shuling Wang
State Key Lab. of Computer Science, Institute of Software
Chinese Academy of Sciences, Beijing, China

Gehang Zhao
School of Mathematical Sciences, Peking University, Beijing, China

Jifeng Hao
Aeronautics Computing Technique Research Institute, Xi'an, China

Hong Ye
Aeronautics Computing Technique Research Institute, Xi'an, China

Bican Xia
School of Mathematical Sciences, Peking University, Beijing, China

ABSTRACT: *A software system normally has several units that have relations with each other. Identifying and measuring such relations is a challenge and a few models have been proposed to do it. Asynchronous models have been used and are executed to using responses. In the current work, we have advocated event-based design for execution. We have listed and explained the event systems and provide calculus for reasoning. The proposed design is applied and used in models in the areas of distributed computing, partition scheduling and real-time operations and protocols.*

Keywords: Hoare Logic, Compositional Verification, Events

Received: 22 September 2022, Revised 20 December 2022, Accepted 26 December 2022

DOI: 10.6025/jcl/2023/14/1/11-29

1. Introduction

In the verification of large-scale computer programs and systems, a major challenge is modular verification: how to verify components of a system independently, and then compose results from verification of each component into an overall correctness result for the entire system. Sometimes, only part of the system is available or needs to be verified, and the question arises of how to properly model the interaction points between the parts to be verified and the rest of the system.

Interaction between components of a computer system comes in many types. The simplest is when one component of the system makes function calls to another component. In this case, if the callee is completely verified, or at least if an abstraction of its behavior is available, then the caller can be verified in terms of the verified or assumed behavior of the callee. Somewhat more complicated is the situation where two or more components make function calls on each other. Verification approaches designed for such situations include assume-guarantee reasoning [1], where during the verification of each component, we make assumptions on the behavior of the components it relies on, and prove guarantees of its own behavior under these assumptions. Correctness of the composed system is then proved by showing that the guarantees of the components entail all of the assumptions.

Another way to describe interactions between components of a system or with the environment is via effects and effect handlers. There is a long line of work on the modeling and verification of effects and effect handlers [4, 24, 30], which will be reviewed in more detail in Section 6. The main idea is to model each interaction as an uninterpreted event that returns a result, and provide a continuation for each possible value of the result, hence modeling the program as an interaction tree [21, 34]. This technique has been applied successfully to the verification of swap servers [21] and an HTTP Key-Value Server [36].

In many applications, interactions between components are of a special kind, where each component sends out events without requiring an immediate response in order to continue with its execution. This can be used, for example, to model asynchronous function calls, commands to other components to carry out some action, or outputs to the environment.

In such cases, interactions can be modeled by lists of output events, together with handlers for such events, which can result in change of state in other components as well as possibly further events. In situations where such a modeling technique is applicable, it provides a simpler and sometimes more accurate way of modeling interaction between components.

In this paper, we propose a framework for modeling and verifying components with such asynchronous interactions, by defining event monads and event systems, as well as a Hoare logic-style calculus for reasoning about them. Using several case studies, we demonstrate that this framework can be applied to a wide range of situations, allowing verification of functional properties in a clear and modular way.

The main motivation for the current work comes from a project for verification of partition scheduling in a commercial real-time operating system implemented following the ARINC 653 standard. The standard requires that the physical resources of the computer are divided into several partitions, and the operating system enforces strong spatial and temporal separation between the partitions. To achieve this, scheduling between partitions is strictly deterministic, based on pre-specified time tables (which however can be switched at run time in response to specific events). As partition scheduling is critical to ensure temporal separation as well as real-time properties of the entire system, it is of strong interest to verify its correctness and precision.

While the scheduling policy based on time tables is itself quite simple, its actual implementation is complicated due to efficiency considerations and the need to support switching between time tables. The implementation involves two components, the scheduler and the watchdog, that interact with each other as well as with other parts of the system. The scheduler adds new tasks with deadlines to the watchdog, and the watchdog invokes the dispatch function of the scheduler when the deadline is reached. The scheduler also receives calls to switch time tables from the environment, and emits calls to change the partition. Likewise, the watchdog must handle tasks with deadlines from other modules. We show that the framework based on event monads can be used to model such bi-directional interactions as well as interactions with the environment, resulting in the modular verification of correctness of partition scheduling based on time tables. The verification is at design-level, with parts of the model closely following the C implementation.

In addition to the application to partition scheduling in real-time operating systems, we also present two smaller case studies, demonstrating the applicability of the framework to other situations. First, we verify a model of distributed computing based on MapReduce. In this model, the client divides a large computing task into several parts, with each part sent to a different server. Each server asynchronously returns the result of the corresponding task after some time. The client then obtains the final result of computation by adding up the answers. Verification of the model requires reasoning about the bi-directional interaction between the client and the servers. Second, we verify a model of cache-coherence protocol.

The protocol to be verified is first proposed by Steven German, and is widely used as a test case in parameterized verification (e.g. in the work of Chou et al. [6]). It involves a number of clients that can obtain either exclusive or shared access to some data, by interacting with the server through request and invalidate messages. We model such interactions using the event monad, and prove that the entire system does guarantee exclusive access when required.

1.1. Implementation in Isabelle/HOL

The work described in this paper is implemented1 in Isabelle/HOL [29]. We base the implementation on the AutoCorres library [12], mainly to take advantage of its wp tactic for verification condition generation. The development of event monads in Isabelle is inspired by, but does not depend logically on the development of nondeterministic state monads in AutoCorres [7]. Nor do we make use of its translation facility from C code.

The rest of this paper will make free use of Isabelle notation. We will just review some frequently-used symbols. ’a × ’b denotes the product of two types, with elements in the form (a, b). Functions fst and snd return the first and second component of a pair. f ‘ S denotes the image of function f on the set S. The symbols @ and # denote append and cons operations on lists, and xs ! i denotes taking the ith element of a list. We will also make frequent use of inductive predicates, using the keyword inductive in Isabelle/HOL, or the version generating sets using inductive_set.

1.2. Outline of the Paper

In Section 2, we motivate the theory in this paper using the example of partition scheduling based on time tables. In Section 3, we define event monads and its associated Hoare logic.

In Section 4, we describe how to combine different components into a single event system, and additional rules for reasoning about event systems. In Section 5, we demonstrate the framework on two smaller examples: MapReduce and cache-coherence protocol, as well as the main application on partition scheduling. We discuss related work in Section 6 and finally conclude in Section 7.

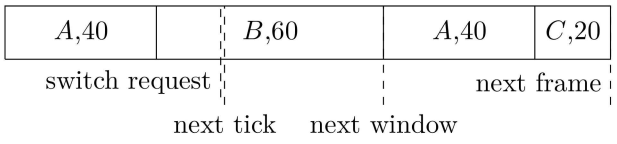
2. Motivation: Partition Scheduling

In this section, we describe the motivating example of this paper: scheduling for a partitioned real-time operating system implementing the ARINC 653 standard.

ARINC 653 is an international standard for real-time operating systems in the aerospace industry [2]. The standard specifies that computing resources are divided into several partitions, with each task running in a single partition. Strong spatial and temporal separation are enforced between partitions, so that failure in one partition will not propagate to affect tasks in other partitions. Part of the mechanism for enforcing temporal separation is a strictly deterministic scheduling policy between partitions based on time tables. A time table specifies the order and allotted time of partition executions in each major time frame. Here time is given in units of ticks. For example, the following time table has a major time frame of 160 ticks and consists of four windows, where partition A executes for the first 40 ticks, partition B executes for the next 60 ticks, partition A executes again for the next 40 ticks, and finally partition C executes for the remaining 20 ticks.

A,40	B,60	A,40	C,20

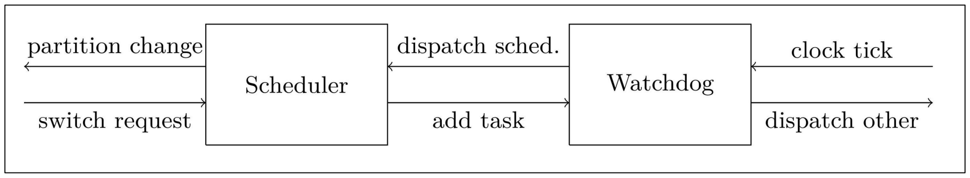
A secondary feature of the scheduling module is the support for switching between time tables. Switching to a new time table can be requested at any time, but is not carried out immediately. Instead the time when the actual switch occurs depends on the switch mode. There are three possible modes: next tick, next window, and next frame. Their meanings are straightforward, and are illustrated in the following diagram.



While the functionality described above is relatively simple, its actual implementation is more complex. This is mainly due to efficiency considerations. Within the operating system, there are many modules that require keeping time. It would be costly to invoke each module that requires time-keeping at every clock tick. Instead, time-keeping is centralized in a single module called the watchdog. Each module can register tasks on the watchdog, each with a specified deadline in the number of ticks. The watchdog will then dispatch each task exactly at its deadline.

Hence, we can consider partition scheduling as an interaction between two components: the scheduler and the watchdog. At initialization, the scheduler sets the partition to that of the first window in the time table, and registers a task to the watchdog with the number of ticks in the first window as deadline. When the task is dispatched, the scheduler sets the partition to that of the next window, and registers a new task to be dispatched after the number of ticks in the next window. This pattern continues, rotating to the first window after reaching the end of the frame. For the time table switch requests, the switch mode and the ID of the next time table are immediately recorded. The switch mode is checked at every dispatch, which will correctly handle the next window and next frame requests. A special check is needed at every clock tick in order to handle the next tick request.

While the scheduler and the watchdog can be viewed as independent modules, there are interactions between them in both directions: the scheduler sends requests to add tasks to the watchdog, while the watchdog dispatches functions in the scheduler. Moreover, both the scheduler and the watchdog interact with other parts of the system: the scheduler receives switch requests, and invokes change of partition. The watchdog receives clock ticks, and may dispatch tasks for other modules. These are illustrated in the following diagram.



All these interactions are of asynchronous nature: they do not require an immediate response in order to continue with the execution. Instead, we can accurately model the system as emitting all interaction events at the end of each function.

As illustration, we give the definition of events in this case study. First, we define some basic datatypes:

```
type_synonym partition = nat
type_synonym ttbl_id = nat
type_synonym task_id = nat
datatype switch_mode = NO_SWITCH | NEXT_TICK | NEXT_WINDOW | NEXT_FRAME | ONGOING
```

The datatype of events is given by

```
datatype event = TICK | DISPATCH nat | SWITCH ttbl_id switch_mode | PARTITION parti-
tion | WATCHDOG_ADD task_id × nat | WATCHDOG_TICK | WATCHDOG_REMOVE task_id
```

Here TICK is the global tick operation, which calls on WATCHDOG_TICK as well as checks whether the current switching mode is next tick. DISPATCH i dispatches the task with index i on the watchdog. We assume this dispatches the scheduler task if i = 0. SWITCH tid mode requests a switch to time table tid under switch mode mode. PARTITION p denotes change of partition to p. The events WATCHDOG_ADD, WATCHDOG_TICK and WATCHDOG_REMOVE corresponds to adding a task, time increment, and removing a task on the watchdog, respectively. The scheduling component handles events DISPATCH 0 and SWITCH, and outputs events PARTITION and WATCHDOG_ADD. The watchdog component handles events

WATCHDOG_ADD, WATCHDOG_TICK and WATCHDOG_REMOVE, and sends DISPATCH events. The environment provides implementation of the TICK events, which may output WATCHDOG_TICK, WATCHDOG_REMOVE, and DISPATCH events.

The overall approach is a compositional verification of the system. First, implementations of the scheduler and the watchdog are specified independently, and we verify their properties (as refinements of functional specifications, including the trace of interaction events). Then, the combined system is specified, and its property (a refinement of an overall functional specification) is verified using results proved about the two components.

3. Event Monad

3.1 Definitions of Event Monads

First, we review the concept of (nondeterministic) state monads, as formalized in Isabelle by Cock et al. in [7]. A state monad over a state of type ’s and returning a value of type ’a is given by the type `’s ⇒ (’a × ’s) set × bool`. Given an input state *s*, it returns a pair (*rs, b*), where *rs* is a set of possible pairs of return value and output state, and *b* is a failure flag indicating whether it is possible for the computation to fail (including non-termination). The bind operation *f* >>= *g* executes f first, then executes *g* applied to the return value of *f*. It can fail if either *f* or *g* can fail.

The event monad is an extension of the state monad, where we also record a trace of events produced by the program. The formalism is parameterized over a type ’s of states and a type ’*e* of events. Then the event monad with return type ’*a* is defined as follows.

```
type_synonym (’s,’e,’a) event_monad = ’s ⇒ (’a×’s×’e list) set × bool
```

Given an input state *s*, the function returns a pair (*rs, b*), where rs is now a set of triples of return value, output state, and trace of interaction events. The meaning of the second component *b* is the same as before.

We begin by defining some basic event monads: skip does nothing, return returns the given value, and signal raises the given event:

```
return a = (λs. ({(a, s, [])}, False))
signal e = (λs. ({((), s, [e])}, False))
skip = return ()
```

The behavior of bind *f* ≫= *g* is similar to that of usual state monads, except the trace of events produced by *g* is appended to the trace produced by *f* to give the overall trace of events. The formal definition is as follows. As preparation, we define *prepend_event* for prepending onto the event trace, and *bind_cont* for applying monad g to an intermediate result of computation:

```
prepend_event e1 (b, s2, e2) = (b, s2, e1 @ e2)
bind_cont g (a, s1, e1) = (let (rs2, f2) = g a s1 in (prepend_event e1 ‘ rs2, f2))
```

Then we define:

```
f ≫= g = (λs. let (rs1, f1) = f s in
              let rss2 = bind_cont g ‘ rs1 in
               (⋃(fst ‘ rss2), f1 ∨ True ∈ (snd ‘ rss2)))
```

Next, we define event monads for retrieving the state, setting the state to *s,* and modifying the state using a function *f.* They are similar to the analogous definitions for state monads.

```
get = (λs. ({(s, s, [])}, False))
put s = (λ_. ({((), s, [])}, False))
modify f = (λs. put (f s))
```

The while loop is defined similarly as in the state monad. Given a loop condition *C* of type `'r ⇒ 's ⇒ bool`, and a loop body of type `'r ⇒ ('s,'e,'r) event_monad`, the expression `whileLoop C B` has type `'r ⇒ ('s,'e,'r)` event_monad. Given an initial value r of type *'r* and state *s*, it repeatedly executes the loop body *B* until the condition *C* becomes false, using the return value of *B* to reset the value of *r* after each iteration. In addition, the traces of events produced by *C* at every iteration are appended in sequence to give the overall trace of events.

Finally, we give the definition of non-deterministic choice between two monads:

```
f ⊔ g = (λs.  let (rs1, f1) = f s; (rs2, f2) = g s in (rs1 ∪ rs2, f1 ∨ f2))
```

3.2 Hoare Logic for Event Monads

We now present a Hoare logic for reasoning about event monads. It will turn out that the definition of Hoare triples as well as the Hoare rules are very similar to that for nondeterministic state monads in [7]. For reference, we repeat these definitions here.

```
⦃P⦄ f ⦃Q⦄ = (∀s. P s ⟶ (∀(r,s') ∈ fst (f s). Q r s'))
no_fail P f = (∀s t. P s t ⟶ ¬(snd (f s)))
⦃P⦄ f ⦃Q⦄! ⟷ ⦃P⦄ f ⦃Q⦄ ∧ no_fail P f
```

For the extension to event monads, the main decision that needs to be made is where to include the dependence on the trace of events. We choose to allow both the precondition and postcondition to depend on traces. The definition for partial correctness in event monads is as follows, and the definition for total correctness is the same as before.

```
⦃P⦄ f ⦃Q⦄ = (∀s es1. P s es1 ⟶ (∀(r,s',es2) ∈ fst (f s). Q r s' (es1 @ es2)))
```

In words, given an initial state *s* and trace *es*1 satisfying precondition *P*, if the execution of *f* gives return value *r*, final state *s*' and trace *es*2, then the *triple r*, *s*' and *es*1 *@ es*2 satisfies the postcondition *Q*. In a sense the trace is viewed as part of the state, with each signal operation appending onto it. The Hoare logic rules have mostly the same form as before. The only new rule is that for signal:

```
⦃λs es. P () s (es @ [e])⦄ signal e ⦃P⦄!
```

As another example, we show the total correctness Hoare triple for the while loop. Note the invariant is a function of the state *s* and the value *r* of variables modified at each iteration of the loop, as well as the current trace *es*, while the loop condition cannot depend on the current trace. Here *wf R* means *R* is a well-founded relation.

```
(⋀s es. P s es ⟹ I r s es) ⟹
(⋀r0 s0. ⦃ λs es. I r0 s es ∧ C r0 s ∧ s = s0 ⦄ B r0
         ⦃ λr' s' es'. I r' s' es' ∧ ((r', s'), (r0, s0) ∈ R ⦄!) ⟹
wf R ⟹ (⋀r s es. I r s es ∧ ¬C r s ⟹ Q r s es) ⟹
⦃P⦄ whileLoop C B R ⦃Q⦄!
```

What distinguishes event monads and its Hoare logic from simply recording the trace within the state is a *frame rule*, which reflects the fact that the trace of events can only be appended onto, not removed or modified. This rule allows us to show for each function only the Hoare triple where the precondition requires the trace to be empty, using the frame rule to cover other cases when necessary.

First, we define the nil and chop assertions on events

```
nil_e = (λes. es = [])
P ^_e Q = (λes. ∃es1 es2. P es1 ∧ Q es2 ∧ es = es1 @ es2)
```

The frame rule is then as follows.

$$\frac{\{\!|\lambda\texttt{s es. P s} \wedge \texttt{nil}_{\texttt{e}}\,\texttt{es}|\!\}\ \texttt{c}\ \{\!|\lambda\texttt{r s es. P' r s} \wedge \texttt{Q s es}|\!\}}{\{\!|\lambda\texttt{s es. P s} \wedge \texttt{R es}|\!\}\ \texttt{c}\ \{\!|\lambda\texttt{r s es. P' r s} \wedge \texttt{(R} \,\hat{}_{\texttt{e}}\, \texttt{Q s) es}|\!\}}$$

An alternative form of the frame rule, more convenient when the list of output events is a function of the initial state, is given as follows.

$$\frac{\{\!|\lambda\texttt{s es. s = s0} \wedge \texttt{es = []}|\!\}\ \texttt{c}\ \{\!|\lambda\texttt{r s t. Q r s} \wedge \texttt{es = g s0}|\!\}}{\{\!|\lambda\texttt{s es. s = s0} \wedge \texttt{es = es0}|\!\}\ \texttt{c}\ \{\!|\lambda\texttt{r s es. Q r es} \wedge \texttt{es = es0 @ g s0}|\!\}}$$

The form of definitions of nil and chop, and the naming of the frame rule will remind the reader of separation logic [31]. However, there are some essential differences: compared to separating conjunction, the chop operator represents joining in the temporal rather than the spatial dimension. It should also be noted that the chop operator is not commutative. In fact our setting is closer to that of interval temporal logic [15] and duration calculus [5].

Remark 1. Another possible choice is to always assume the trace to be empty in the precondition, and allow only the postcondition to depend on the trace, perhaps also separating it into two assertions, on the state and trace respectively. However, with this choice, the Hoare rules will no longer be in weakest precondition form, making verification condition generation more difficult. We also note that our approach allows postconditions to state relationships between the final state and the trace (e.g. there exists *n* such that the value of variable *x* is *n* and the additional trace contains exactly *n* events).

4. Event System

With event monads, we can specify and verify properties of programs that produce a trace of events. However, what makes events truly useful is in combination with event handlers. In this section, we define the concept of event system as a model of reactive system consisting of event monads, and a Hoare logic-style calculus for reasoning about event systems.

4.1. Definition of Event System

An event system models a reactive system as a partial mapping from events to their handlers, which take the form of event monads with the same event type, and no return value:

```
type_synonym ('e,'s) event_system = 'e ⇒ ('s,'e,unit) event_monad option
```

We now define the execution of an event *e* in event system *sys*. The intuitive idea is as follows. If *e* is not handled by *sys*, then it is simply output to the environment. Otherwise, the event monad handling e is executed. Suppose the resulting trace of events is *es*, then each event in *es* is recursively executed in sequence.

The formal definition is given by two inductive predicates defined by mutual recursion. The predicate *reachable sys e s (s', es')* means starting from state s, executing event e can reach state *s*' and output event trace *es*' to the environment. The predicate *reachable_list sys es s (s', es')* is similar, except *es* is a list of events to be executed in sequence. Note the output trace *es'* does not include events that are handled within the system.

```
reachable_None: sys e = None ⟹ reachable sys e s (s, [e])
reachable_Some: ⟦sys e = Some p; (r, s', es) ∈ fst (p s); ¬snd (p s);
            reachable_list sys es s' (s'', es')⟧ ⟹ reachable sys e s (s'', es')


reachable_list_Nil: reachable_list sys [] s (s, [])
reachable_list_Cons:⟦reachable sys e s(s', es1);reachable_list sys es s'(s'', es2)⟧
  ⟹ reachable_list sys (e # es) s (s'', es1 @ es2)
```

We also define (by a similar induction) the concept of guaranteed termination when executing an event e or a sequence of events es starting from state *s*. These are written as *terminates sys e s and terminates_list sys es s*.

```
terminates_None: sys e = None ⟹ terminates sys e s
terminates_Some: sys e = Some p ⟹ ¬snd (p s) ⟹
   (∀(r,s',es') ∈ fst (p s). terminates_list sys es' s') ⟹ terminates sys e s
```

```
terminates_list_Nil: terminates_list sys [] s
terminates_list_Cons: terminates sys e s ⟹
   (∀s' es'. reachable sys e s (s', es') ⟶ terminates_list sys es s') ⟹
   terminates_list sys (e # es) s
```

4.2 Hoare Logic for Event Systems

Based on the Hoare logic for event monads, we present a Hoare logic-style calculus for reasoning about event systems. Since handlers for events are assumed to have no return values, both pre- and post-conditions are predicates on the pair of state and event trace only. We define partial and total correctness of an event system for a single event and a sequence of events as follows.

```
sValid sys P e Q =
 (∀s es1 s' es2.  P s es1 ⟶ reachable sys e s (s', es2) ⟶ Q s' (es1 @ es2))
sNo_fail sys P e = (∀s es.  P s es ⟶ terminates sys e s)
sValidNF sys P e Q ⟷ sValid sys P e Q ∧ sNo_fail sys P e
```

```
sValid_list sys P es Q =
 (∀s es1 s' es2.  P s es1 ⟶ reachable_list sys es s(s', es2) ⟶ Q s'(es1 @ es2))
sNo_fail_list sys P es = (∀s es'.  P s es' ⟶ terminates_list sys es s)
sValidNF_list sys P es Q ⟷ sValid_list sys P es Q ∧ sNo_fail_list sys P es
```

We now state rules for deriving Hoare triples for event systems. The rules are stated for total correctness only, with the partial correctness case being similar. First, if an event e is not handled by the system, it just appends to the trace:

$$\frac{\texttt{sys e = None}}{\texttt{sValidNF sys } (\lambda \texttt{s t. P s (t @ [e])) e P}} \quad \text{SYS-NONE}$$

The central rule concerns the case where e is handled by the system. It makes use of the Hoare triple for the corresponding event monad c. To show the overall postcondition R, we need to show for each intermediate state s and event trace es that is allowed by the postcondition of c, that R is satisfied after executing es starting from s. This is expressed in the following:

$$\frac{\begin{array}{c}\texttt{sys e = Some c} \\ \{\!|\lambda \texttt{s es. P s} \wedge \texttt{nil}_e \texttt{ es}|\!\}\ \texttt{c}\ \{\!|\lambda \texttt{r s es. Q s es}|\!\}! \\ \bigwedge \texttt{s es. Q s es} \Longrightarrow \texttt{sValidNF\_list sys } (\lambda \texttt{s' es'. s' = s} \wedge \texttt{nil}_e \texttt{ es'})\texttt{ es R}\end{array}}{\texttt{sValidNF sys } (\lambda \texttt{s es. P s} \wedge \texttt{nil}_e \texttt{ es) e R}} \quad \text{SYS-SOME}$$

This rule only considers the case where the initial trace is empty. The frame rule (to be described below) is then used for the general case.

The two rules for event lists are mostly straightforward:

$$\texttt{sValidNF\_list sys P [] P} \quad \text{SYS-NIL}$$

$$\frac{\texttt{sValidNF sys P e Q} \quad \texttt{sValidNF\_list sys Q es R}}{\texttt{sValidNF\_list sys P (e \# es) R}} \quad \text{SYS-CONS}$$

We can then prove Hoare triples for general lists of events using induction rules in Isabelle.

For example, the following rule is proved by induction on the length of es, for the case where none of the events in es is handled by the system:

$$\frac{\forall e \in \texttt{set es. sys e = None}}{\texttt{sValidNF\_list sys } (\lambda \texttt{s t. P s (t @ es)) es P}} \quad \text{SYS-ALL-NONE}$$

Another useful rule concerns append of event lists, stated simply as follows:

$$\frac{\texttt{sValidNF\_list sys P es1 Q} \quad \texttt{sValidNF\_list sys Q es2 R}}{\texttt{sValidNF\_list sys P (es1 @ es2) R}} \quad \text{SYS-APPEND}$$

In addition to the above rules, there are also the usual rules for weakening the precondition, strengthening the postcondition, and dealing with the logical operations. We omit the details here.

Both partial and total correctness Hoare triples for event systems satisfy frame rules. Here we give the rules for total correctness and a single event. The rules for partial correctness and for a sequence of events are similar.

$$\frac{\texttt{sValidNF sys } (\lambda \texttt{s t. P s} \wedge \texttt{nil}_e \texttt{ t) e } (\lambda \texttt{s t. P' s} \wedge \texttt{Q s t)}}{\texttt{sValidNF sys } (\lambda \texttt{s t. P s} \wedge \texttt{R t) e } (\lambda \texttt{s t. P' s} \wedge \texttt{(R } \hat{}_e \texttt{ Q s) t)}}$$

To verify properties of an event system, we first prove appropriate Hoare triples for each event handler as event monads. Then, these results are composed together using the above rules. Often, there is a logical order among events handled by the system, so that for each event handler, any output event that is handled occurs earlier in the order. In this case, the sValid and sValidNF statements can be proved in sequence following this logical order. This will be demonstrated in Section 5.3.2 (where the order is WATCHDOG_ADD, DISPATCH 0, WATCHDOG_TICK). In more general cases induction techniques would be needed to show several sValid and sValidNF statements at the same time.

4.3 Composition of Event Systems
Usually, event systems are composed of multiple subsystems, with most of the events acting on some of the subsystems only. In the case studies in Section 5, we will compose subsystems by pairing, as well as using parameterized array of subsystems. We provide support for this by defining functions that automatically transform monads acting on subsystems to monads acting on the entire state, as well as Hoare rules for dealing with monads defined in this way.

First, we consider composition of subsystems by pairing. Given two subsystems with state 's and 't respectively, we can form a new system with state 's × 't. Monads acting on 's and 't can be transformed into monads acting on the global system using the following functions:

```
apply_fst_st t (r, s, es) = (r, (s, t), es)
apply_fst c (s, t) = (let (rs, f) = c s in ((apply_fst_st t) ‘ rs, f))

apply_snd_st s (r, t, es) = (r, (s, t), es)
apply_snd c (s, t) = (let (rs, f) = c t in ((apply_snd_st s) ‘ rs, f))
```

Given a Hoare triple for program c, we automatically get a Hoare triple for program apply_fst c or apply_snd c, using the following Hoare rules (the rules for apply_snd and for total correctness are similar).

$$\frac{\{\!|\lambda \texttt{s es. P s es}|\!\} \texttt{ c } \{\!|\lambda \texttt{r s es. Q r s es}|\!\}}{\{\!|\lambda \texttt{p es. P (fst p) es} \wedge \texttt{R (snd p)}|\!\} \texttt{ apply\_fst c } \{\!|\lambda \texttt{r p es. Q r (fst p) es} \wedge \texttt{R (snd p)}|\!\}}$$

Another common way of composing systems is via parameterized array. This is used in both case studies on MapReduce and cache-coherence protocols. We begin by defining a function to transform a monad to apply on the ith index of an array:

```
apply_idx_st slist i (r, s, es) = (r, slist[i := s], es)
apply_idx c i slist = (let (rs, f) = c (slist ! i) in ((apply_idx_st slist i) ‘ rs, f))
```

For reasoning rules about apply_idx c i, we provide two versions. The first version is well-suited to the case where the behavior of c is deterministic, giving by some function f. Then the behavior of apply_idx c is simply applying f to the ith index of the array:

```
⋀s0. ⦃λs es. s = s0 ∧ es = []⦄ c ⦃λ_ s es. s = f s0 ∧ es = g s0⦄
--------------------------------------------------------------------
  ⦃λs es. s = slist ∧ es = []⦄ apply_idx c i
  ⦃λ_ p es. s = slist[i := f (slist ! i)] ∧ es = g (slist ! i)⦄
```

The second version is better suited to the case where c is characterized by properties on the initial and final states, and there is a certain uniformity between properties satisfied by s ! i for each index i. This is used, for example, in the verification of MapReduce, where each server satisfies some uniform property parameterized by the data it contains.

```
      ⋀i. i < N ⟹ ⦃λs es. P i s ∧ nil_e t⦄ c ⦃λr s es. Q i r s es⦄  j < N
-------------------------------------------------------------------------------
 ⦃λs es. length s = N ∧ (∀i. i < N ⟶ P i (s ! i)) ∧ nil_e es⦄ apply_idx c j
 ⦃λr s es. length s = N ∧ (∀i. i < N ⟶ ¬i = j ⟶ P i (s ! i) ∧ Q j r (s ! j) es⦄
```

5. Case Study

In this section, we present three case studies applying the above framework to different scenarios. Two smaller case studies concern distributed computing based on MapReduce, and a cache-coherence protocol proposed by Steven German. The main case study concerns partition scheduling using time tables in a real-time operating system. In all of the case studies, we show that the use of event monads allow us to separately specify and verify each component of the system. The specifications can then be composed together to form an overall correctness result.

5.1 MapReduce

MapReduce is a method of distributed computing proposed by Dean and Ghemawat in [8]. The idea is to divide a large computation task into several smaller portions, each portion consisting of applying some function f (the map stage). The results are then combined together by applying another function g onto the initial value and each returned result in sequence (the reduce stage). We demonstrate the verification of MapReduce using a simple example, which nevertheless contains the main ingredients, including asynchronous communication and nondeterminism in the time when each machine returns its result.

Given a number *N* and a list of lists of numbers data of length *N*, we need to compute the total sum of numbers in data. This is done using *N* servers, where each server *i* computes the sum of `data ! i`. The results are then collected together in a client.

The state of each server is as follows:

```
record server =
  input :: nat list
  index :: nat
  cursum :: nat
  returned :: bool
```

Here input is the list of numbers whose sum is to be computed. `index` indicates the current progress of computation, and cursum records the sum of numbers between 0 and `index - 1`. Finally, returned indicates whether the computation is complete, with sum already returned to the client.

The state of the (unique) client is as follows:

```
record client =
  num_received :: nat
  acc :: nat
  alldone :: bool
```

Here num_received indicates the number of returned results received from the servers. acc is the accumulated value of the sum, and alldone indicates whether the entire computation has finished. Note the client does not record which servers it has received results from (hence it does not know the sum of which lists the value acc corresponds to), which presents additional challenges to verifying correctness of the system.

The events of the system are defined using the following datatype.

```
datatype event = QUERY nat × nat list | RESPOND nat | TICK | INIT
```

Handlers of the events are as follows. QUERY is handled by the server, and initializes its state:

```
query_impl xs = (put (⦇input = xs, index = 0, cursum = 0, returned = False⦈))
```

The event TICK applies the following monad to each server node. As long as the node has not returned its answer, it nondeterministically chooses to perform a step, which amounts to either progress the computation by one index, or returning the result when reaching the end.

```
tick_node_impl = (get ⪢= (λs.
  (if ¬returned s then
     if index s = length (input s) then
       signal (RESPOND (cursum s)) ⪢= (λ_. put (s⦇returned := True⦈))
     else put (s⦇index := index s + 1, cursum := cursum s + input s ! (index s)⦈)
   else skip) ⊔ skip))

tick_impl N = (whileLoop (λi _. i < N)
  (λi. apply_fst (apply_idx tick_node_impl i) ⪢= (λ_. return (Suc i))) 0) ⪢=
  (λ_. return ())
```

Response is handled by the client, and applies the following monad. It updates the number of received answers and the currently accumulated sum. Moreover, if the number of received answers reaches N, it sets the alldone flag.

```
respond_impl N a = (get ⪢= (λs.
  put (s⦇acc := acc s + a, num_received := num_received s + 1⦈)) ⪢= (λ_.
  get ⪢= (λs. if num_received s = N then put (s⦇alldone := True⦈) else skip)))
```

The handler for INIT uses a while loop to send data to all server nodes:

```
init_impl N data = (whileLoop (λi _. i < N)
  (λi. signal (QUERY (i, data ! i)) ⪢= (λ_. return (Suc i))) 0) ⪢= (λ_. return ())
```

Finally, we define the event system with global state given by server list × client as follows:

```
fun system :: nat⇒nat list list⇒(event, server list×client) event_system where
  system N data (QUERY (i, xs)) = Some (apply_fst (apply_idx (query_impl xs) i))
| system N data (RESPOND a) = Some (apply_snd (respond_impl N a))
| system N data TICK = Some (tick_impl N)
| system N data INIT = Some (apply_snd (init_impl N data))
```

The condition on the initial state is:

```
init_state N (ss, c) ⟷
  length ss = N ∧ c = ⦇num_received = 0, acc = 0, alldone = False⦈
```

We prove the correctness result that starting from the state satisfying init_state, after performing one INIT and any number of TICK events, if the client has set the alldone flag, then the value of acc in the client must be the total sum of data. This is stated formally as follows:

```
N ≥ 0 ⟹ sValidNF_list (system N data)
  (λp es. init_state N p ∧ nilₑ es)
  (INIT # replicate n TICK)
  (λp es. (alldone (snd p) ⟶ acc (snd p) = sum (λi. sum_list (data ! i)) {0 ..< N})
          ∧ nilₑ es)
```

Since whether a step is performed is nondeterministic according to the definition of `tick_node_impl`, it is impossible to predict how many TICKs are needed for the alldone flag to be set. The same property would hold if a more detailed specification about when to perform a step is used. The main idea of the proof is to verify that TICK preserves an invariant of the system, stating that each server node has either returned or is in progress of computing the sum, and the values of num_received and acc in the client correctly keeps track of the number and total sum of the returned answers. The proof makes key use of the second rule for `apply_idx` given in Section 4.3, lifting the property for each server to a property for the collection of servers.

5.2 Cache-Coherence Protocol

Our second example concerns a cache-coherence protocol proposed by Steven German, which has been widely used as a test case for parameterized verification [6]. The protocol consists of one server and multiple client nodes, and is intended to enforce either exclusive or shared access to some data. If a client node requires exclusive (resp. shared) access, it sends a ReqE (resp. ReqS) message to the server. On receiving a ReqE message, the server sends invalidation messages Inv to all client nodes that currently have exclusive or shared access. On receiving an invalidation message, the client sets its own state to Invalid and returns an InvAck message back to the server. On receiving InvAck messages from all clients with access, the server sends an SendE message to the client node that initially requested access. On receiving an SendE message, the client knows that it now has exclusive access, and sets its own state to Exclusive. The handling of ReqS is similar, except there is no need to send invalidation messages if no node has exclusive access, and the server sends SendS message to the client that initially requested access, who then sets its own state to Shared.

Hence, the interaction is mediated by six types of events, defined as follows:

```
datatype event = ReqS nat | ReqE nat | Inv nat | InvAck nat | SendS nat | SendE nat
record server =
invset :: bool list
shrset :: bool list
curptr :: nat option
grantE :: bool
```

Here invset records which client nodes the server is waiting for InvAck from. shrset indicates which client nodes currently have exclusive or shared access. curptr stores the currently requesting client node, and grantE indicates whether the requested access is exclusive. The state of the client is simply `record client = st :: state`, where `st` is one of `Invalid, Shared, or Exclusive`.

In this model, `Inv, InvAck, SendS, and SendE` are all generated by event handlers in either server or clients, whereas `ReqS and ReqE` can be seen as events coming from the environment.

Correctness of the system can be stated as an invariant that is preserved by the `ReqS and ReqE` events, and which implies exclusive access when required. This can be stated for `ReqS` in the following theorem (the one for `ReqE` is similar).

```
 i < N ⟹ sValidNF (system N)
```

```
(λp es. system_inv N p ∧ nil_e es)
(ReqS i)
(λp es. system_inv N p ∧ nil_e es)
```

where `system_inv` contains a number of conditions, including the following:

```
∀i<N. (st (clist ! i) = Shared ∨ st (clist ! i) = Exclusive) ⟶
       (shrset s ! i ∨ invset s ! i)       \
∀i<N. ¬invset s0 ! i
grantE s0 ⟶ (∀i j. i < N ⟶ j < N ⟶ i = j ⟶ ¬shrset s0 ! i ∨ ¬shrset s0 ! j)
```

Together, they state that a client node can be in shared or exclusive state only if the corresponding bit in the shrset or invset array is turned on. However, before and after each ReqE and ReqS event, none of the invset is turned on, while in the case when grantE equals true, at most one shrset is turned on. Hence in this case at most one client node has exclusive access.

5.3. Partition Scheduling

We now describe the application of our framework to verify partition scheduling using time tables. We perform two versions of verification: with and without allowing switching between time tables. The version without switching already contains interaction between the scheduler and watchdog in both directions, hence illustrates the main ideas of the framework. The version with switching shows scalability to examples of moderate complexity. Verification of the watchdog is shared between the two versions, and will be described first below.

5.3.1 Watchdog

The watchdog module maintains a set of tasks, each with its own deadline. Tasks are indexed by elements of type task_id. Abstractly, the state of a watchdog can be represented by a partial mapping from task ID to deadline (in the number of clock ticks):

```
type_synonym astate_watchdog = task_id ⇒ nat option
```

Concretely, a watchdog is implemented as a doubly-linked list (the watchdog chain), where each node represents a task, consisting of task ID and deadline relative to the previous task in the chain. The deadline at the first node of the chain is the actual deadline. The advantage of recording relative deadline is that at each tick, only the deadline at the head of the chain needs to be updated. In this paper, we model the watchdog chain as an array, which preserves most of the logical complexity, without requiring reasoning about linked lists. Hence, the type of concrete watchdog is given by

```
type_synonym watchdog_chain = (task_id × nat) list
```

For example, the watchdog chain (1, 10), (2, 5), (4, 0), (3, 2) means the task with ID 1 is due in 10 ticks, Two tasks with IDs 2 and 4 are due in 15 ticks, and a task with ID 3 is due in 17 ticks.

We define `rel_w` as the refinement relation between abstract and concrete watchdog representations as follows:

```
valid_watchdog es ⟷
 (length es > 0 ⟶ snd (es ! 0) > 0) ∧ (∀evt_id. occurs_atmost_one es evt_id)
event_time [] i = None
event_time (e # es) i =
 (if fst e = i then Some (snd e)
  else case event_time es i of None ⇒ None | Some k ⇒ Some (k + snd e))
rel_w aw cw ⟷ valid_watchdog cw ∧ aw = event_time cw
```

Here *`event_time cw i`* returns None if *i* is not in the watchdog chain `cw`, and `Some k` if it is in the chain with actual `deadline k`. The condition valid_watchdog contains the invariant that the first deadline in the chain is always positive, and each task appears at most once in the chain.

The concrete watchdog operations are implemented as follows. Adding a new task requires traversing the chain, inserting the task at the correct location, and decrement the deadline of the next node accordingly. The tick operation first decrements the deadline at the head of the chain by 1, then removes all tasks from the head that have zero deadlines, and emitting their dispatch events. The remove operation first locates the task to be removed in the chain, removes it, then increments the deadline of the next node accordingly.

It is nontrivial to verify the correctness of the watchdog module (see the statistics in Section 5.4). This is stated in terms of refinement between abstract and concrete specifications, including correctness of the list of DISPATCH events emitted when handling `WATCHDOG_TICK`, as well as correct update of the watchdog chain when handling `WATCHDOG_ADD` and `WATCHDOG_REMOVE`.

5.3.2 Scheduler with No Switching

We now describe the verification of scheduler without considering switching between time tables. The abstract state of the scheduler consists of the time table (which stays unchanged), and the ID of the current window:

```
record astate_scheduler =
 as_ttbl :: time_table
 window_id :: nat
```

Dispatch increments window_id by one (modulo the total number of windows) and produces two output events: change of partition (which is output to the environment in the combined system) and adding to the watchdog (which is handled by the watchdog module).

The concrete state, which corresponds more closely to the actual implementation in C, maintains in addition the length of the current window `(window_time)` and the amount of time passed in the current frame `(cur_frame_time)`. The variable `window_id` in the abstract state is renamed to `cur_window`:

```
record cstate_scheduler =
 cs_ttbl :: time_table
 window_time :: nat
 cur_window :: nat
 cur_frame_time :: nat
```

The invariant to be maintained is that `window_time` is the length of the window with index cur_window, and cur_frame_time is the total length up to (but not including) `cur_frame_time`. During dispatch, first increment `cur_frame_time` by window_time; if the result equals the length of the major frame, then cur_window and `cur_frame_time` are reset to zero, otherwise cur_window is incremented by one; finally `window_time` is updated, and the events PARTITION and WATCHDOG_ADD are emitted. Again, correctness of the scheduler is stated and proved as refinement between the abstract and concrete specifications. The refinement relation rel_s requires that `window_id` in the abstract state equals `cur_window` in the concrete state, and the invariant to be maintained held for the concrete state.

5.3.3 Combined System

The state of the combined system is the product of concrete states for the scheduler and the watchdog. The event handlers are defined in terms of handlers in the two subsystems:

```
sys (DISPATCH 0) = Some (apply_fst dispatch_impl)
sys (WATCHDOG_ADD (ev, n)) = Some (apply_snd (wadd_impl (ev, n)))
sys WATCHDOG_TICK = Some (apply_snd wtick_impl)
```

The refinement relation in the product system is the product of the refinement relations on the two sides:

```
rel_total (as, aw) p ⟷ rel_s as (fst p) ∧ rel_w aw (snd p)
rel_total (as, aw) p ⟷ rel_s as (fst p) ∧ rel_w aw (snd p)
```

We then verify the overall system specifications stated as Hoare triples in the event system, following the method described at the end of Section 4.2. We briefly describe the proved specifications. WATCHDOG_ADD adds a new task with given deadline to the watchdog chain, without producing additional events. DISPATCH 0 outputs a PARTITION event to indicate change of partition, as well as adding task 0 with a new deadline back to the watchdog chain.

The input event WATCHDOG_TICK is the main entry point of the system. Its handler produces a list of DISPATCH i events for all tasks $i \neq 0$ that should be dispatched at the current step, as well as performing the action of DISPATCH 0 if task 0 should be dispatched.

5.3.4 Top-Level Refinement

Based on the results proved in the previous section, it is possible to prove a more abstract specification of the combined system, stating that the scheduling follows the time table precisely. For this, we first define an overall abstract state as follows.

```
record astate =
 a_ttbl :: time_table
 frame_time :: nat
 wchain :: astate_watchdog
```

The state contains the constant time table (`a_ttbl`), the number of ticks spent in the current frame (`frame_time`), and the watchdog mapping excluding the scheduler task (`wchain`). The functional specifications `spec_atick and spec_atick_ev` (omitted here) state that frame_time increments by 1 at each tick (modulo frame length), change of partition is emitted only at window boundaries, and the usual dispatch events for other tasks are emitted at appropriate times. Then the theorem stating the top-level refinement is:

```
sValidNF sys
 (λp es. arel_full a p ∧ nil_e es)
  WATCHDOG_TICK
 (λp es. arel_full (spec_atick a) p ∧ distinct es ∧ set es = spec_atick_ev a)
```

Description	Files	Number of lines
Foundations	EventSpecWhile, EventSystem	1131
MapReduce	MapReduce	726
Cache-coherence	Cache	1261
Time table	TimeTable	270
Watchdog	Watchdog, EventSystemWatchdog	2572
Scheduler (no switch)	EventSystemScheduler	723
Scheduler (switch)	EventSystemSwitchScheduler	866
	EventSystemSwitch	1399
Total		8948

Table 1. Statistics of the implementation and examples

Here `arel_full` is the refinement relation between `astate` and the pair of abstract states (*as, aw*) in the previous section. The theorem is proved directly from the previous result for `WATCHDOG_TICK`, by proving the refinement between the specification in the previous section and the specifications `spec_atick and spec_atick_ev`.

5.3.5 Scheduler with Switching

We also verified a version of the scheduler allowing switching between time tables, which forms a more substantial example showing the scalability of our framework. For reason of space, we only sketch the main additional features:

• There is an additional input event `SWITCH n mode`, which requests switching to a new time table with identifier n, with switch mode given by mode.

• The dispatch function in the scheduler tests for two of the switch modes: next window and next frame, and performs switching at the appropriate time.

• There is an event for overall clock tick which handles the next tick switch mode. If next tick is active, the handler sets the mode to next window, then emits three events: remove the scheduler task from watchdog chain, perform watchdog tick, and dispatch the scheduler. Otherwise, it simply emits the event to perform watchdog tick.

As with the case with no switching, we combined correctness of the scheduler and the watchdog to form correctness result of the overall system. It states that scheduling proceeds precisely according to the time table, and whenever a switch event arrives, the switch to a new time table will be performed at the appropriate time according to the switch mode. The correctness theorem is again stated in the form of a refinement between the concrete behavior of the system and an abstract functional specification.

5.4 Statistics

Statistics from the implementation of the framework and the examples are given in Table 1. Definingx the event monad and event system, then setting up the Hoare logic take around 1000 lines in total. Verifying the watchdog is surprisingly complex, so it is a good thing that under the framework, it only need to be done once for verifying the two versions of the scheduler.

6. Related work

There is a large body of work on effects and effect handlers [4, 24, 30], game semantics [20], and verification methods using interaction trees [21, 34, 36], which study how to model and verify interacting systems where interactions are synchronous, and a response is needed immediately in order to proceed with execution of the program. Hence, programs are modeled using interaction trees which branch at every continuation in response to an event. Compared to these works, we consider a different model, which may be simpler or more accurate in some settings, where sending events is asynchronous. Closely related is the work of Ahman et al. on asynchronous effects [3]. However, they focused mostly on type checking of programs with asynchronous effects, rather than verification of functional correctness.

Zhao et al. [39] proposed a framework for rely-guarantee reasoning about reactive systems defined by events. In this work (and many earlier works that specify systems using events), each event has a guard, and is triggered whenever the guard is satisfied. The semantics in our case is quite different, where events are triggered explicitly, either by the environment or by other event handlers. In this respect, our semantics is closer to that of I/O automata [26, 27], but with sequential rather than concurrent execution.

There is a large number of frameworks for program verification in Isabelle and other proof assistants, many of which based on monads and/or refinement. We will only give some examples in Isabelle here. Our work builds upon the state monads of Cock et al. [7], which are used extensively in the seL4 project [19]. Lammich et al. developed the Isabelle Refinement Framework [23, 22], which was most recently used by Haslbeck and Lammich for verification of functional correctness and worst-case complexity of algorithms at the LLVM level [16]. Tuong et al. developed Clean [32], which implements a state-exception monad in Isabelle, and is used to verify a number of small programs. Foster et al. developed Isabelle/UTP [11], implementing Hoare and He's Unifying Theories of Programming [17], and applied it to the verification of reactive and hybrid systems [10, 9].

There have been many existing work on verifying operating systems or its components [13, 18, 19, 35, 37]. This includes much work on the specification and verification of separation kernels [40]. Zhao et al. [38] specified channel-based communication according to the ARINC 653 standard, and provided formal proofs about its information flow security. Verbeek et al. [33] formalized the API specification for PikeOS, and proved security properties as required by the MILS architecture. Murray et al. also extended the verification of the seL4 microkernel to prove information flow enforcement properties [28]. More recently, there have been focus on verification of the scheduling in real-time operating systems. In particular, the work of Guo et al. [14] and Liu et al. [25] verified the correctness of scheduling in a real-time version of CertiKOS. They verify both the correct implementation of a scheduling policy as well as schedulability under that policy.

7. Conclusion

In this paper, we introduced a framework for modular verification of interacting components using event monads. Procedures in each component are modeled by event monads which can produce a trace of events. Each event can then be handled by procedures in other components in the event system. We applied the framework to the verification of three examples, including partition scheduling based on time tables in a real-time operating system. These indicate that the framework is applicable in a wide range of scenarios.

While the current paper focuses on verification of distributed systems and operating system components, it appears likely that verification using event monads and event systems can also be applied in other contexts, such as network communication. Exploring these other applications is a goal of future work.

Referennces

[1] Abadi, M. & Lamport, L. (1993) Composing specifications. *ACM Transactions on Programming Languages and Systems*, 15, 73–132 [DOI: 10.1145/151646.151649].

[2] Aeronautical Radio, Inc. & ARINC (2015). Specification 653: Avionics Application Software Standard Interface, Part 1–Required Services. *ARINC Airlines Electronic Engineering Committee*.

[3] Ahman, D. & Pretnar, M. (2021) Asynchronous effects. *Proceedings of the ACM on Programming Languages. Proceedings of the ACM Program*. Lang, 5, 1–28 [DOI: 10.1145/3434305].

[4] Bauer, A. & Pretnar, M. (2015) Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming. Algebraic Methods Program, 84, 108–123* [DOI: 10.1016/j.jlamp.2014.02.001].

[5] Chaochen, Z., Hoare, C.A.R. & Ravn, A.P. (1991) A calculus of durations. Information Processing Letters, 40, 269–276 [DOI: 10.1016/0020-0190(91)90122-X].

[6] Chou, C.-T., Mannava, P.K. & Park, S. (2004) A simple method for parameterized verification of cache coherence protocols. *In: Formal Methods in Computer-Aided Design, 5th International Conference*, FMCAD 2004, Austin, TX, USA, 15–17 November, 2004, Proceedings, pp. 382–398.

[7] Cock, D., Klein, G. & Sewell, T. (2008) Secure microkernels, state monads and scalable refinement. *In: Theorem Proving in Higher Order Logics, 21st International Conference*, TPHOLs 2008, Montreal, Canada, 18–21 August, 2008. Proceedings, pp. 167–182 [DOI: 10.1007/978-3-540-71067-7_16].

[8] Dean, J. & Ghemawat, S. (2008) MapReduce: Simplified data processing on large clusters. Communications of the ACM, 51, 107–113 [DOI: 10.1145/1327452.1327492].

[9] Foster, S. (2019), Proceedings Hybrid relations in isabelle/utp. In: Unifying Theories of Programming - 7th International Symposium, UTP 2019, Dedicated to Tony Hoare on the Occasion of His 85th Birthday, Vol. 8. October: Porto, Portugal, pp. 130–153.

[10] Foster, S., Baxter, J., Cavalcanti, A., Miyazawa, A. & Woodcock, J. (2018) Automating verification of state machines with reactive designs and isabelle/utp. *In: Proceedings Formal Aspects of Component Software* - 15th International Conference, FACS 2018, Pohang, South Korea, 10–12 October, 2018, pp. 137–155.

[11] Foster, S., Baxter, J., Cavalcanti, A., Woodcock, J. & Zeyda, F. (2020) Unifying semantic foundations for automated verification tools in Isabelle/UTP. *Science of Computer Programming*, 197, 102510 [DOI: 10.1016/j.scico.2020.102510].

[12] Greenaway, D. (2014). *Automated Proof-Producing Abstraction of C Code. URL: handle.unsw.edu.au/1959.4/54260 [PhD Thesis]*. University of New South Wales: Sydney.

[13] Gu, R., Shao, Z., Chen, H., Wu, Xiongnan (N), Kim, J., Sjöberg, V. & Costanzo, D. (2016) Certikos: An extensible architecture for building certified concurrent OS kernels. *In: 12th USENIX Symposium on Operating Systems Design and Implementation*, OSDI 2016, Savannah, GA, USA, 2–4 November, Vol. 2016. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gu, pp. 653–669.

[14] Guo, X., Lesourd, M., Liu, Mengqi, Rieg, L. & Shao, Z. (2019) Integrating formal schedulability analysis into a verified OS kernel. *In: Proceedings, Part II Computer Aided Verification* - 31st International Conference, CAV 2019, New York City, USA, 15–18 July, 2019, pp. 496–514 [DOI: 10.1007/978-3-030-25543-5_28].

[15] Halpern, J.Y., Manna, Z. & Moszkowski, B.C. (1983) A hardware semantics based on temporal intervals. *In: Automata, Languages and Programming, 10th Colloquium, Barcelona, Spain. Proceedings. ITP Foundation* 2022, 1983, 278–291 [DOI: 10.1007/BFb0036915].

[16] Maximilian P. L. Haslbeck and Peter Lammich. For a few dollars more - verified fine-grained algorithm analysis down to LLVM. In Programming Languages and Systems - 30th European Symposium on Programming, ESOP 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, pages 292–319, 2021.

[17] C.A.R Hoare and Jifeng He. Unifying Theories of Programming. Prentice-Hall, 1998.

[18] Gerwin Klein. *Operating system verification–an overview*. Sadhana, 34:27–69, 2009.

[19] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby C. Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an OS microkernel. *ACM Trans. Comput. Syst*., 32(1): 2:1–2:70, 2014. doi:10.1145/2560537.

[20] Jérémie Koenig and Zhong Shao. Refinement-based game semantics for certified abstraction layers. *In LICS '20: 35th Annual ACM/IEEE Symposium on Logic in Computer Science*, Saarbrücken, Germany, July 8-11, 2020, pages 633–647, 2020. doi:10.1145/3373718.3394799.

[21] Nicolas Koh, Yao Li, Yishuai Li, Li-yao Xia, Lennart Beringer, Wolf Honoré, William Mansky, Benjamin C. Pierce, and Steve Zdancewic. From C to interaction trees: specifying, verifying, and testing a networked server. In Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019, pages 234–248, 2019. doi:10.1145/3293880.3294106.

[22] Peter Lammich. Refinement to imperative HOL. *J. Autom. Reason.*, 62(4):481–503, 2019.

[23] Peter Lammich and Thomas Tuerk. Applying data refinement for monadic programs to hopcroft's algorithm. *In Interactive Theorem Proving* - Third International Conference, ITP 2012, Princeton, NJ, USA, August 13-15, 2012. Proceedings, pages 166–182, 2012.

[24] Thomas Letan, Yann Régis-Gianas, Pierre Chifflier, and Guillaume Hiet. Modular verification of programs with effects and effects handlers. *Formal Aspects Comput*., 33(1):127–150, 2021. doi:10.1007/s00165-020-00523-2.

[25] Mengqi Liu, Lionel Rieg, Zhong Shao, Ronghui Gu, David Costanzo, Jung-Eun Kim, and Man-Ki Yoon. Virtual timeline: a formal abstraction for verifying preemptive schedulers with temporal isolation. *Proc. ACM Program. Lang*., 4(POPL):20:1–20:31, 2020. doi:10.1145/ 3371088.

[26] Nancy A. Lynch and Eugene W. Stark. A proof of the Kahn principle for input/output automata. Inf. Comput., 82(1):81–92, 1989. doi:10.1016/0890-5401(89)90066-7.

[27] Nancy A. Lynch and Mark R. Tuttle. Hierarchical correctness proofs for distributed algorithms. *In Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, Vancouver, British Columbia, Canada, August 10-12, 1987, pages 137–151, 1987. doi:10. 1145/41840.41852.

[28] Toby C. Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao,

and Gerwin Klein. sel4: From general purpose to a proof of information flow enforcement. In 2013 *IEEE Symposium on Security and Privacy*, SP 2013, Berkeley, CA, USA, May 19-22, 2013, pages 415–429, 2013.

[29] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. Isabelle/HOL - *A Proof Assistant for Higher-Order Logic*, volume 2283 of Lecture Notes in Computer Science. Springer, 2002. doi:10.1007/3-540-45949-9.

[30] Gordon D. Plotkin and Matija Pretnar. *Handlers of algebraic effects. In Programming Languages and Systems, 18th European Symposium on Programming*, ESOP 2009, York, UK, March 22-29, 2009. Proceedings, pages 80–94, 2009. doi:10.1007/978-3-642-00590-9_7.

[31] John C. Reynolds. Separation logic: A logic for shared mutable data structures. *In 17th IEEE Symposium on Logic in Computer Science (LICS 2002)*, 22-25 July 2002, Copenhagen, Denmark, Proceedings, pages 55–74, 2002. doi:10.1109/LICS.2002.1029817.

[32] Frédéric Tuong and Burkhart Wolff. Clean - an abstract imperative programming language and its theory. Arch. Formal Proofs, 2019, 2019. URL: https://www.isa-afp.org/entries/ Clean.html. B. Zhan, Y. Lv, S. Wang, G. Zhao, J. Hao, H. Ye, and B. Xia 33:21

[33] Freek Verbeek, Oto Havle, Julien Schmaltz, Sergey Tverdyshev, Holger Blasum, Bruno Langenstein,Werner Stephan, BurkhartWolff, and Yakoub Nemouchi. *Formal API specification of the pikeos separation kernel. In NASA Formal Methods - 7th International Symposium*, NFM 2015, Pasadena, CA, USA, April 27-29, 2015, Proceedings, pages 375–389, 2015.

[34] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. Interaction trees: representing recursive and impure programs in Coq. Proc. ACM Program. Lang., 4(POPL):51:1–51:32, 2020. doi:10.1145/3371119.

[35] Fengwei Xu, Ming Fu, Xinyu Feng, Xiaoran Zhang, Hui Zhang, and Zhaohui Li. *A practical verification framework for preemptive OS kernels*. In Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II, pages 59–79, 2016. doi:10.1007/978-3-319-41540-6_4.

[36] Hengchu Zhang, Wolf Honoré, Nicolas Koh, Yao Li, Yishuai Li, Li-yao Xia, Lennart Beringer, William Mansky, Benjamin C. Pierce, and Steve Zdancewic. *Verifying an HTTP key-value server with interaction trees and VST. In 12th International Conference on Interactive Theorem Proving*, ITP 2021, June 29 to July 1, 2021, Rome, Italy (Virtual Conference), pages 32:1–32:19, 2021.

[37] Yongwang Zhao and David Sanán. Rely-guarantee reasoning about concurrent memory management in Zephyr RTOS. *In Computer Aided Verification* - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II, pages 515–533, 2019. doi:10.1007/978-3-030-25543-5_29.

[38] Yongwang Zhao, David Sanán, Fuyuan Zhang, and Yang Liu. Reasoning about information flow security of separation kernels with channel-based communication. *In Tools and Algorithms for the Construction and Analysis of Systems* - 22nd International Conference, TACAS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings, pages 791–810, 2016.

[39] Yongwang Zhao, David Sanán, Fuyuan Zhang, and Yang Liu. A parametric rely-guarantee reasoning framework for concurrent reactive systems. In Formal Methods - The Next 30 Years - Third World Congress, FM 2019, Porto, Portugal, October 7-11, 2019, Proceedings, pages 161–178, 2019. doi:10.1007/978-3-030-30942-8_11.

[40] Yongwang Zhao, Zhibin Yang, and Dianfu Ma. A survey on formal specification and verification of separation kernels. *Frontiers Comput. Sci.*, 11(4):585–607, 2017. doi: 10.1007/s11704-016-4226-2.