VERIFYING NETWORKS OF PROCESSES THAT SYNCHRONIZE VIA SHARED VARIABLES

Tatjana Kapus, Bogomir Horvat

Abstract

An approach to modelling and verification for networks of processes that synchronize solely via shared variables is presented. A process or a network is defined by its set of possible behaviours. Each behaviour is an abstraction of an infinite execution sequence of the process. The specification of a network can be obtained naturally from the specifications of its component processes. It is possible to verify liveness properties by using standard temporal logic operators.

1 Introduction

An active area of research in the last decade is modelling, formal specification and verification of algorithms that exhibit concurrent behaviour. The present research has been motivated by the following. An important class of concurrent algorithms is mutual exclusion algorithms. Their purpose is to ensure that two processes are not in their critical sections simultaneously. Solutions to the mutual exclusion problem typically employ synchronization via shared variables. Concurrent executions of processes may be seen as interleavings of their atomic actions or events. By synchronization, some constraints on the ordering of events are achieved (e.g., constraints on the ordering of entries to and exits from critical sections) [11]. A number of approaches to verification of mutual exclusion algorithms and of networks of processes synchronizing via shared variables in general exist. Verification by model checking (e.g., [5, 15]) or related approaches ([1]) can be quite straightforward. They allow fully automated, algorithmic verification of finite-state systems, but it bothers that the shared variables have to be modelled like any other process. Another disadvantage is that the complete state space of the system has to be constructed in order to prove a property which might be proved in a few steps by an axiomatic approach.

This paper presents an axiomatic approach for verifying networks of processes that execute concurrently, asynchronously, and synchronize exclusively via shared variables. We propose a model and provide a temporal logic formalism for specification and verification of their safety and liveness properties. The approach enables us to form the specification of a network naturally from component process specifications.

2 The model

A process is determined by its name, a set of variables, and a set of events in which it may engage. Note that we are only concerned with proving properties strictly regarding synchronization, and not, for example, with functional correctness. Therefore, in this model, we are interested in variables, events and their meaning purely to the extent to which they are essential to model problems of synchronization via shared variables.

We will use the following notation for sequences of elements. A (possibly infinite) sequence s of elements is written as σ₀, σ₁, σ₂, ... For example, s₀ refers to the first value of s. σ₀ refers to the subsequence beginning at s₁, i.e. σ₀ = s₁, s₂, ... The length of sequence s is denoted by |s|. · is used as an infix catenation symbol. s = t means that s is a prefix of sequence t.

2.1 Events and variables

Formally, let P be the process name, V be the finite set of variables, and E be the finite set of events or the alphabet of P. Each variable v ∈ V has a function type(v) that associates with it a range called the type of v. We partition V into two disjoint subsets: V₁ = V₁ ∪ V₁. V₁ is the set of variables which can share with other processes and V₁ is the set of "hidden" variables. V₁ is the set of variables which cannot share with other processes ("nonhidden" variables).

E is partitioned into two disjoint subsets, such that E = E₁ ∪ E₁. Here, E₁ is the set of independent events, and E₁ is the set of dependent events of process P. The occurrence of independent events does not depend (directly) on any variable of the process. The occurrence of dependent events depends on values of some variables from V₁.

For every process P, its process description has the form:

P(V₁ : T₁, V₁ : T₁, E₁, E₁).

Let T₁ be the set of types associated with variables from V₁, T₁ = {type(v)v ∈ V₁}. Similarly, T₁ = {type(v)v ∈ V₁}. By T₁, V₁, and V₁ : T₁ we denote the set of all pairs v : type(v), such that type(v) ∈ T₁ and v ∈ V₁, or type(v) ∈ T₁ and v ∈ V₁, respectively.

Any of the four sets or just the sets of types may be omitted from a process description if they are clear from the context.

Both dependent and independent events may be of a special kind - assignments of values to the variables from V₁ or V₁. Thus, also E = A ∪ A. A is the set of assignment events. Every assignment event e ∈ A only has a name. Each assignment event e ∈ A, on the contrary, is of the form a(v, k). a is the name of e. v ∈ V₁ is the variable to which a value is to be assigned, and k is the value. The value must be in accordance with the type of v. k ∈ type(v). By var(e) we refer to the variable of e, i.e. var(a(v, k)) = v.

A process is either a primitive process or a network - parallel composition of finite number of processes. A primitive process has no hidden variables.

We view a primitive process as executing some internal computations and communicating via a shared variable from time to time with a process that has the same variable to share in order to synchronize with it. Internal computations are represented by
nonassignment events. We are not interested in what kind of internal computation a nonassignment event means. It can even represent an assignment to a local variable. Because we do not model local variables, this does not matter in our model. For an event, its nonassignment character only means that it does not represent an assignment to a shared variable. The so-called assignment events represent communication of the process with its environment via shared variables. They model writing to the variables. Reading from them is modelled implicitly by dependent events. We will later show that a dependent event cannot happen unless certain enabling condition holds on the value of a shared variable. If the occurrence of an action is completely independent, or if it depends only on the occurrence of some independent or dependent events, the action is modelled by an independent event.

Events are indivisible and have no duration. A time-consuming action has to be modelled by a pair of events, the first denoting its start and the second representing its finish.

If we compose the primitive process with a process which has the same shared variable as the former, we obtain a network. Of course, we can compose processes containing no shared variables, as well. The shared variable is a shared variable of the network, too. In our model, more than two processes may synchronize via the same shared variable. Thereby, the network may further be composed with a process which contains this variable so as to synchronize through it. By hiding the variable, we prevent such a composition because synchronization of the network with any other process via the hidden variable is not allowed.

2.2 Observations and behaviours

A process is characterized by its set of behaviours. A behaviour is an infinite sequence of observations on events and variables of the process.

Observations can be treated as states through which the process evolves as time progresses. An observation records the trace of events of the process that have happened up to the moment of the observation, current values of variables of the process, and current truth values of enabling conditions for each dependent event of the process. A trace of a process is defined as a finite sequence of symbols recording the events in which the process has engaged up to some moment in time [7]. In fact, an observation records the trace of a process.

An observation is a generalization of trace. It records all the data necessary to reason about process executions.

Formally, an observation on set $E = E_{dep} \cup E_{seq}$ of events and set $V = V_E \cup V_s = \{v_1, v_2, \ldots, v_n\}$ is a triple

$$(t, Val, Cond).$$

$t$ is a trace on $E$. $Val$ is a n-tuple $(v_1, \ldots, v_n)$ of values of variables from $V$. In fact, we should write $V = \{\text{type}(v_1), \ldots, \text{type}(v_n)\}$ for $V$ contains variable names. By $v_i$ we denote the value of $v_i$ in the observation, such that $v_i \in \text{type}(v_i)$, for $i = 1, \ldots, n$.

Any way, it is not necessary to write "\text{type}(v_1)" because the meaning of $v_i$ is always clear from the context. $Cond$ is a function from $E_{dep}$ to \{true, false\}.

$Cond$ is an enabling condition function. $Cond(e)$ means the occurrence of event $e$ is enabled.

A behaviour on $E$ and $V$ is an infinite sequence

$$\sigma = (t_0, Val_0, Cond_0), (t_1, Val_1, Cond_1), \ldots$$

of observations on $E$ and $V$, such that:

(1) $t_0$ is empty.

(2) For all $n \geq 0$, $t_{n+1}$ equals $t_n$ or is an extension of it, i.e. is $t_n$ followed by some event $e \in E$. In the latter case, if $e \in E_{dep}$, then $Cond_0(e)$ is true. And if $e \in A$ (where either $e \in E_{seq}$ or $e \in E_{dep}$) is of the form $(v, k)$ for some $v \in V$ and $k \in \text{type}(v)$), then $v_{n+1} = k$.

(3) For all $n \geq 0$ and for all "$v" \in V_s$, $v_{n+1} \neq v_n$ only if $t_{n+1} = t_n \cdot e$, for some $e \in A$, such that $\text{var}(e) = "v".$

These are the basic properties of behaviours.

(1) says that initially, the trace is empty because no events have happened yet.

From (2) it is evident that concurrency is modelled by interleaving. The next trace extends the current trace by at most one event. The trace also cannot become shorter as time progresses. This statement in (2) determines that a dependent event cannot happen if its enabling condition in the previous observation is not satisfied. The effect of an assignment on a variable is also specified. Note that the variable gets a new value in the same observation as the assignment event happens. Thus, the atomicity of writing to variables is modelled. The requirement in (3) prevents the hidden variables of a process from being affected by any other process.

The idea of behaviours as infinite sequences of observations comes from Nguyen et al. [13]. They have constructed a model and temporal proof system for networks of processes that communicate exclusively by message-passing. A process can be characterized by its traces (e.g. [7, 12]), but it is difficult to specify liveness properties in trace-based models. This is a consequence of the fact that traces are finite sequences, while some liveness properties (e.g. a progress property that eventually something will happen) hold only on infinite execution sequences. One way to make it possible to specify temporal properties is modelling process computations by infinite traces or sequences of states [4, 9, 10]. Even if specification of liveness properties in a model is possible, it is not necessary that the model allows compositional verification of them (e.g. [9]). Besides, although an approach allows compositional verification of temporal properties, their specification and verification can be complicated (e.g. [3]).

Introduction of observations as generalizations of traces and behaviours into the model allows better specification of liveness (progress) properties. By the model the authors have achieved a compositional proof system not only for safety, but also for liveness properties.

Nguyen et al. require that behaviours of a process be closed under finite repetition of observations. A set $B$ of behaviours is closed under finite repetition if for any two behaviours $\sigma$ and $\sigma'$ the following holds: if $\sigma'$ can be obtained from $\sigma$ by repeating a (possibly infinite) number of observations, each finitely many times, then $\sigma \in B$ if and only if $\sigma' \in B$. Any set of behaviours has a closure under finite repetition. Closure under finite repetition allows them to model concurrent events by nondeterministic interleaving of sequential events without causing interference. This also has the consequence that the notion of "time" is a qualitative one. Time is abstracted to total ordering, and one cannot talk about the exact time at which an event occurs.

Although the Nguyen's model is a message-passing and dataflow model, we make the same requirements for behaviours, because they are useful in our model, too. Safety and progress may be specified in a similar way. Properties of networks can be proved from the component specifications by similar simple proof rules. Of course, the structure of observations differs from those in [13]. In message-passing systems, it is necessary to observe communications on input and output channels of processes, and enabling conditions of communication events. Since we want to
specify and reason about synchronization of events through the communication via shared variables, observations in our model record occurrences of assignment and nonassignment events, their enabling conditions and current values of shared variables. In contrast to [13], we are not concerned about information-hiding, or achieving external specifications of processes, i.e. descriptions of behaviours as observed on their interfaces. In case of message-passing, process interfaces are communication channels, and interfaces in shared variable models are shared variables [3, 14]. Note that we allow internal events in our model. We only want to verify properties relating to synchronization of events. It would not be sensible not to talk in terms of internal events if they are of primary concern (6).

3 The specification and verification formalism

By verification of an algorithm we mean the following. First, we write precise specifications of its component processes in our model. Parallel composition of the processes represents the algorithm in the model. Then we specify properties the algorithm should satisfy. In order to prove that it satisfies them, we have to prove that the properties follow from the assertions obtained by parallel composition of the components.

3.1 Temporal logic specifications

We use standard propositional logic operators and (linear-time) temporal logic future operators for writing assertions on behaviours - specifications. Temporal operators may be used because behaviours are infinite sequences of observations which can be interpreted as states. Infinite sequences of states are models of linear-time temporal logic. We use the following temporal operators: $\Box$ (necessarily), $\Diamond$ (always), $\varnothing$ (eventually), $\mathcal{U}$ (until), $\mathcal{N}$ (unless), $P$ (precedes), and axioms and inference rules of temporal logic as defined in [9, 10].

Since our model is based on the same idea as the Nguyen’s one, our specification language and that one associated with the Nguyen’s model are very much alike. For this reason, we will not define our language completely in this paper. The details can be found in [9, 13]. The set of basic symbols in the language (individual constants and variables, predicate and function symbols) is partitioned into two subsets: global symbols and local symbols. The global symbols have a uniform interpretation and maintain their values or meanings from one state to another. Quantification is allowed over global variables only as in [13]. The local symbols may assume different values in different states of the sequence. Local function and predicate symbols in the assertion language are allowed, too.

A model $(I, \alpha, \sigma)$ for our language consists of a global interpretation $I$, a global assignment $\alpha$, and an infinite sequence of states $\sigma = \sigma_0, \sigma_1, \ldots$. Interpretation $I$ specifies a nonempty domain $D$ and assigns concrete elements, functions and predicates to the global individual constants, function and predicate symbols. Assignment $\alpha$ assigns a value to each global free variable. Each state of $\sigma$ is an assignment of values to the local free individual variables, function and predicate symbols. Definitions of truth values for temporal logic formulae and terms can be found in [13]. Here we provide two examples.

Let $w_1^*, I$ being implicitly assumed, denote the truth value of a temporal formula or term $w$.

If $w$ does not contain any modal operators, then $w_1^*$ is the value of $w$ in $\sigma_0$ under the assignment $\alpha$.

As an example of temporal formula, the truth value of the formula $\Box w$ is defined as follows:

$$w_1^* = true$$

Informally, $\Box w$ may be read as "eventually $w$ will be true". If $w$ is a term, then $w_1^*$ denotes the value of $w$ in next moment.

We say that a model satisfies $w$ whenever $w$ is true in the model.

In fact, the alphabet of a process contains event symbols which represent different event classes [7]. Let $e$ be an event symbol, and $t$ a trace on a set of event symbols including $e$. We will use $e$ to denote the restriction of $t$ to symbol $e$ in temporal assertions. By restricting $t$ to $e$ we obtain a trace containing exactly the occurrences of events $e$ in $t$. So, $|e|$ means the number of occurrences of $e$ in $t$. We use event symbols as local variables in our language. They are called the sequence - of - events variables because they denote sequences of these events. The variable symbols from the set of variables of the process are also used as local variables. They are called the variable value variables. The local function symbol used is $\mathcal{C}(e)$ corresponding to the enabling condition function.

To find out whether a behaviour satisfies a temporal assertion, we have to know how an observation $s$ is treated as a state. We have to assign the sequence of events $e$ in the trace of a to each local sequence-of-events variable $e$. To each local variable value free variable $e$ its value in the observation $s$ has to be assigned. Also, the corresponding enabling condition function has to be assigned to the local function symbol $\mathcal{C}(e)$.

Like in [13], we have to assign to the local predicate symbol $\mathcal{C}(e)$ the "precedes" relation on the events of the trace of the observation: $\mathcal{C}(e)\mathcal{C}(f)$ if the $m$th event $e$ occurs before the $n$th event $f$ in the trace.

$\mathcal{C}(e)$ is a total ordering.

A specification of a process $P(V_e : T_e, V_h : T_h, End_e, End_h)$ has the form $\{PV_e : T_e, V_h : T_h, End_e, End_h\}$. $R$ is a temporal assertion in which the only local free variables are event symbols from $End_e$, and variable symbols from $V_e$ and $V_h$. The only local function symbol in $R$ is $\mathcal{C}(e)$. The only local predicate symbol is $\mathcal{C}(e)$, which is needed only to axiomatize behaviours. For any $e \in End_e$, $\mathcal{C}(e)$ may not occur in $R$.

$(P)R$ means: Every behaviour of $P$ satisfies $R$.

We said before that we write precise specifications of component processes when we want to verify a network. Specification $(P)R$ is precise if: every behaviour on $P$'s variables and events is a behaviour of $P$ if it satisfies $R$.

The next operator may not be used for specifying processes in our model because we model concurrent execution of events by interleaving. Behaviours specified by using $\Box$ would not be closed under finite repetition. Besides, we should not say that an event will happen in the next moment when we are not interested in the exact time of events [8]. We need the next operator for axiomatization of the basic properties of behaviours only.

As an example, let us write a precise specification of the following process $P$: $P$ may synchronize with other processes via Boolean variables $A$ and $B$. Initially, they are set to false. Eventually, the process sets $A$ to true. Afterwards it executes action act1. It then waits until $B$ gets true, and executes act2.

Here is a variant of the specification:

$$\{P(\{A, B : Boolean\}, \{\{ass(A, true), act1\}, \{act2\})\} S$$

where
The standard model of temporal logic is an infinite sequence of states determined by current values of variables [9, 14], or an infinite sequence of events [4]. In that case one would write Oe to specify that event e happens eventually. Note that we cannot write assertions on single events in our model, but always on sequences of events. This is due to the fact that the observations, i.e. states in a behaviour, contain traces, and not single events. For example, to specify that eventually a process will execute an event e, we assert that the number of events e in the process trace (denoted by |e|) will eventually be for 1 greater than the number in the current observation. Safety and liveness properties may be specified and verified separately as in [13].

Process P has nonhidden variables A and B, independent assignment event ass(A, true) and independent nonassignment event act1, and dependent event act2. The first five lines contain safety specifications. The remaining conjuncts are liveness specifications. One must be careful when specifying that an event will eventually happen. For example, it would be wrong if we wrote OCond(act2) in the precedent of the last conjunct instead of OCond(act2). We can say that the specification of P defines so-called open behaviours of the process [14]. For a process with nonhidden variables, it can be specified, how and when it changes them. But we do not know exactly, how the variables will change in time, because they are "open" to be changed by the environment. When writing the specification S, we had to take into account that future could change B at liberty all the time. It is too strong a requirement for a dependent event to happen eventually once its condition is true. The requirement that it must happen if it is enabled infinitely often is weak enough. It represents weak fairness [10], and can be compared with the fairness assumption for synchronous communication between processes in [13].

In precise specifications, it is not allowed to write explicit assertions (e.g. C = 4) on the values of variables in the process, except for initial observation. Note that the basic properties of behaviours say nothing about the initial values of variables of the process. Every precise specification must contain initialization assertions for all variables of the process. Thus, the values of variables in the initial observation of the process are determined. The first line of S in our example contains initialization assertions. The temporal formulae that express properties we want to prove from precise specifications need not contain initialization assertions. They also may contain explicit statements about the values of variables for any observation.

3.2 Axioms for behaviours and proof rules

To work formally with specifications, we need axioms and inference rules describing the domain of values in which process variables may range, and those relating to sequences. We also need axioms and inference rules for temporal logic, and axioms that define the basic properties of behaviours. Proof rules are needed to derive specifications of networks built from primitive processes given together with their precise specifications.

We have already described the basic properties of behaviours informally. With the approach presented in this paper (as in [13]), specifications are written by assuming that these properties already hold for every behaviour (see e.g. the previous example of precise specification). This means that we write assertions as if those specifying the basic properties have already been written. To prove any properties about behaviours formally in the temporal language, the basic properties have to be stated in temporal logic as axioms. Here we give the axioms:

(a) |e| = 0, where e is a sequence-of-events variable, i.e. the initial trace is empty.

(b) O(0 ≤ |O e| ≤ |e| + ... + |O e| = |e| ≤ 1), for n = 0, 1, ..., where e0, e1, ... are (local) sequence-of-events variables, i.e. the next trace extends the current trace by at most one element.

(c) O(e ⊤ O = Cond(e)) for dependent events e, i.e. a dependent event can occur only if its enabling condition holds.

(d) O(a(v, k) κ O(a(v, k)), for a hidden variable v and corresponding assignment events named a, i.e. a hidden variable can change the value only as the consequence of occurrence of an assignment event referring to the variable.

(e) O(v ⊤ O = a(v, k) κ O(a(v, k)), for a hidden variable v and corresponding assignment events named a, i.e. a hidden variable can change the value only as the consequence of occurrence of an assignment event referring to the variable.

(f) ∀m, n = (e, m) ⇐ (f, n)), i.e. the ordering among the elements of a trace is preserved as the trace is extended.

From axioms (b), (f), and (g) it follows that O(e ⊢ e).

Finally, we provide the proof rules.

Network formation rule:

\[((P_i \mid V_{i1} \mid V_{i2} \mid V_{i3}) \cup \mid V_{i4} \mid V_{i5} \cup V_{i6} = \{i_1, i_2, 2, 2, i_3 \mid j_1; E_{i1} \cup E_{i2} \cap (E_{i1} \cup E_{i3}) = \{i_1, i_2, 2, 2, i_3 \mid j_1; E_{i1} \cup E_{i2} \cup E_{i3} \cup E_{i4} \cup E_{i5} \cup E_{i6} \cup E_{i7} \}) \mid R_i \land R_j)\]

The network formation rule says that parallel composition of processes satisfies conjunction of their specifications. In order for parallel composition to be sensible, the initialization assertions of component processes must not contradict each other. Their events and variables must be properly named. The proper naming can be achieved by conventional uniform substitution of names in logical formulæ of specifications.

If a component process contains a hidden variable, the other process may not contain the same variable (either hidden or nonhidden). This requirement, together with axiom (e) for behaviours, guarantees two things. Firstly, the hidden variable will never be changed, except by the process which owns it. And secondly, it may not be read by the other process, so that the latter cannot in any way synchronize via it. Note that component processes may contain the same nonhidden variable. By their parallel composition, it becomes a nonhidden variable of the network. The processes in the network may synchronize via it with each other, and by the environment. A hidden variable of a component process becomes a hidden variable of the network, too. The independent events of components are independent events of the network, and the same holds for dependent events.
From the network formation rule it is evident why we must not assert that something will happen in next moment. If we composed two process specifications, the first one stating that an event would happen in the next moment, and the second one stating the same thing about another event, we could not prove anything about the network. The conjunction would amount to false because of axiom (b) about the interleaving in behaviours.

A general rule for writing the precise specification of a process is that we may not assert anything that could contradict with its environment.

**Hiding rule:**

\[
(P(V, V'))R; H \subseteq V,
\]

\[
((P \land H)(V' - H, V' \cup H)) R \land \forall v \in H \Xi_v.
\]

Axiom (c) for behaviours states that a hidden variable can be changed only by an appropriate assignment event. We call the axiom a hiding assumption for the variable. The hiding assumption for variable \( v \) is denoted by \( \Xi_v \). The hiding rule says that by hiding some nonhidden variables of process \( P \) specified by \( R \), we obtain a process which satisfies conjunction of \( R \) and of hiding assumptions for each newly hidden variable.

Let us suppose that we would like to prove properties about an algorithm, which consists of two processes synchronizing via a common nonhidden variable. In their parallel composition, the variable is still nonhidden. It means that it may be changed freely by other processes, but such behaviours are not legal behaviours of the algorithm. Only the two component processes should be allowed to change it. Therefore, in order to verify the algorithm, we have to hide it. The axiom for hiding is called an assumption because it is not necessary to write it in the specification. It suffices to move the newly hidden variable from the set of nonhidden variables into the set of hidden variables in the process description. This already means that the hiding assumption has to be considered.

Here is a more concrete example of hiding. If we leave variable \( A \) in process \( P \) from the example of precise specification nonhidden, we can only prove that eventually its value will be true as the consequence of the assignment named \( \text{ass} \). We do not know whether or not it will remain true afterwards. If we hide it, only \( P \) is allowed to change it. From \( S \) and from \( \Xi_v \) we can prove that once it gets true, it will remain so forever.

**Consequence rule:**

\[
(P)R, R \Rightarrow S
\]

\[
(P)S
\]

The consequence rule, together with the hiding rule, is used to prove different properties of networks of processes out of the precise network specifications.

## 4 Conclusion

We have presented a temporal logic approach to verification of networks in which processes synchronize via shared variables. There have been attempts to specify and verify temporal properties of concurrent algorithms without temporal logic (e.g. [1, 2]) because it can be difficult to use. In our model, writing specifications may seem more difficult than in standard approaches ([9, 10, 14]) for two reasons. The first one is that the next operator must not be used. And secondly, because some properties hold in advance for every behaviour, one might be unsure about, what else should be specified. Besides, many specifications can be written almost without temporal operators since much may be expressed only by constraints on the number of events in traces.

To make specification and verification easier, we should write specifications very systematically, and, in our opinion, not too redundantly. We have used the approach to prove that a variant of Peterson's mutual exclusion algorithm [2] guarantees mutual exclusion of two processes, and that eventually a process enters its critical section if it requires its execution.

## References


