Compositional Temporal Logic Based on Partial Order

Adrianna Alexander Wolfgang Reisig

Humboldt-Universität zu Berlin
Unter den Linden 6
10099 Berlin, Germany

alexander@informatik.hu-berlin.de *reisig@informatik.hu-berlin.de*

Abstract

The Temporal Logic of Distributed Actions (TLDA) is a new temporal logic designed for the specification and verification of distributed systems. The logic supports a compositional design of systems: subsystems can be specified separately and then be integrated into one system. TLDA can be syntactically viewed as an extension of TLA. We propose a different semantical model based on partial order which increases the expressiveness of the logic.

1. Introduction

Temporal logic has established itself as an appropriate tool for describing and proving properties of distributed systems. The idea of specifying a system as a logical formula was first proposed by Pnueli [16]. This means that all the possible courses of actions (or states) of the a system are exactly the models of the formula.

A *property* of a system will also be represented as a logical formula. Thus, no formal distinction will be made between a system and a property. Hence, proving that a system possess a property is reduced to proving a logical implication. This is a fundamental benefit of this approach.

There are also some other considerable advantages: Compositional reasoning will be eased significantly: Large systems are composed of smaller components. Properties of the composed system should be derivable from the properties of its components. The components are represented as logical formulas. It can be shown that *parallel composition* of the components can basically be represented by *conjunction* of the formulas representing the components. Furthermore, it is often desirable to express that a higher-level system is *implemented* by a lower-level one. This can simply be represented in logical terms by *implication*.

However, on the other hand, to describe composition of concurrent systems in temporal logic is not a simple task (cp. [6]):

A system is usually described by help of *variables*: A variable updates its value in the course of time. To represent a possible system execution, one usually assumes temporal snapshots of the actual values of the variables in a system. Such a snapshot is most often called a *global state*, formally a mapping $state : Var \to Val$, with *Var* the set of variables and *Val* the set of (potential) values of the system. A sequence $s_0\ s_1\ s_2\ s_3 \ldots$ of global states is called a *state sequence* of a system. Each pair $s_i s_{i+1}$ of adjacent global states forms a *step*.

We specify such a system S with a temporal formula, Φ, whose models are exactly the state sequences of S. Φ will be called a *specification* of S. Suppose now that we wish to use the system S as a part of some modular systems in which other components are working in parallel. A state sequence of the composed system might possibly involve steps at much more frequent intervals than a state sequence of S. Hence, the values of the variables under the control of S are not updated during these intermediate steps (for the sake of simplicity suppose that the variables of S are unaffected by the other components). Thus, there is no guarantee that the state sequences of the composed system will still be models of Φ. Consequently, the composed system cannot be specified by conjunction of Φ and the formulas representing the other components.

A similar problem arises with implementation.

There are several solutions to these problems [6, 8, 9, 15, 12]. In this paper, we are considering one of them: *stuttering invariance*. Lamport's Temporal Logic of Actions (TLA, [12, 3]) is based on this idea (another example is the Modular Temporal Logic, MTL [15]).

In this approach a temporal formula representing a system will be forced to be stuttering invariant, i.e. its truth is preserved under addition or removal of a finite number of repetitions of the same state in a state sequence. As a result of this syntactic restriction a specification formula of S remains true even though another system is running in parallel

Proceedings of the 11th International Symposium on Temporal Representation and Reasoning (TIME'04)
1530-1311/04 $20.00 © 2004 **IEEE**

with S. This makes it actually possible to specify composition of concurrent systems as conjunction. On the other hand, however, stuttering causes some undesirable effects which will be described in more detail in Section 2.

As a solution for the composition (and implementation) problem described above we suggest a new temporal logic, called Temporal Logic of Distributed Actions (TLDA). TLDA is syntactically similar to TLA, but has a semantic model different from that of TLA, called a *run*, which is based on a *partial order*. A run consists of two components: Firstly, the *history* of each variable, i.e. the sequence of its *updates*, secondly, the *synchronization* of updates. This extends the information provided by sequences of global states.

Hence, the composition and implementation between systems can be specified in our logic as conjunction and implication, respectively (Section 5). Furthermore, due to the partial order based semantic model we can explicitly distinguish between *concurrent* and the *nondeterministic* variable updates (see [5]). Moreover, it can be determined, whether an update of a variable does not change the value of the variable or whether the variable is not updated at all (Section 4).

2. Motivation

We start with a simple example, which demonstrates a problem arising from stuttering and justifies our approach.

2.1. A Problem with Stuttering

Let M be a system with three variables x, y and z. There are two actions in M, A_1 and A_2, which are performed nondeterministically: A_1 swaps the values of x and y, A_2 reads the (current) value of y and assigns to z the value $z+y+1$. The variables x, y and z have initially the values 1, 1 and 0, respectively.

Additionally, assume a (weak) *fairness* requirement for M stating that every action eventually continuously enabled in the system would also be infinitely often executed. Both actions, A_1 and A_2, are continuously enabled in M. Thus, both of them have to be executed infinitely often.

As an example, $A_1A_2A_1A_1A_2\ldots$ is an execution of M satisfying the fairness requirement. This execution generates the following *fair* state sequence of M:

$$\begin{array}{lcccccccc} x\colon & 1 & 1 & 1 & 1 & 1 & 1 & \ldots \\ y\colon & 1 & 1 & 1 & 1 & 1 & 1 & \ldots \\ z\colon & 0 & 0 & 2 & 2 & 2 & 4 & \ldots \end{array} \qquad (1)$$

In contrast, the execution $A_2A_2A_2\ldots$ violates the fairness requirement, since the action A_1 is not performed at all. Thus, the generated state sequence

$$\begin{array}{lcccc} x\colon & 1 & 1 & 1 & \ldots \\ y\colon & 1 & 1 & 1 & \ldots \\ z\colon & 0 & 2 & 4 & \ldots \end{array} \qquad (2)$$

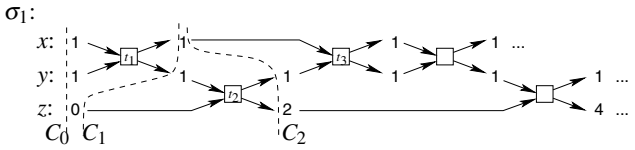
is intuitively *unfair* and is therefore to be excluded from the set of sequences representing the system M. Hence, (1) and (2) should be distinguishable in a formalism used for specifying M.

Unfortunately, they can not be distinguished in a formalism based on stuttering sequences like TLA, since sequences, in which a finite number of iterations of the same global state is added or removed, are equivalent. Consequently, an action changing no variable values, like A_1 in the above example, can not be described. This implies that we cannot detect whether or not such an action is treated fair in a computation. Note, that we could detect this for the action A_1 in case the initial values of x and y happened not to be equal.

This type of actions, however, is quite common in programming languages and it seems reasonable to expect that they could be described in a specification formalism. [17] addresses this problem as causing troubles when specifying semantics of rewriting languages.

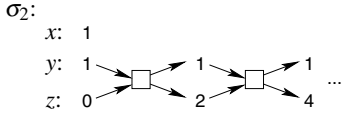
2.2. A Partial Order Solution

We suggest a different model instead. We represent a (distributed) system *not* as a set of sequences of global states, but as a set of partially ordered sets of local updates, called runs. The updates of a single variable are obviously totally ordered. Updates of different variables are partially ordered: They occur in a run either *concurrently* or are sometime enforced to occur *coincidently*. This principle depicts reality more faithfully than stuttering. We will depict each update of a variable explicitly as a box. In our example, the action A_1 coincidently updates the variables x and y and the action A_2 coincidently updates the variables y and z. Hence,



(3)

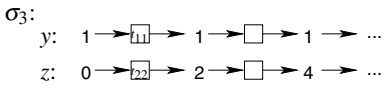
is the run representing the execution $A_1;A_2;A_1;A_1;A_2\ldots$ (the labels $C_0\ldots C_2$ and $t_1\ldots t_3$ will be explained in Section 3.1) and



(4)

the run representing $A_2;A_2;A_2\ldots$. Obviously, there are no concurrent updates in the system M. An example of a run of another system, in which all y-updates would be concurrent

to the z-updates is shown in (5).



(5)

For the sake of compositionality, we assume, as TLA does, that the set of variables *Var* of a system as well as the set of values *Val* are infinite. Hence, we will explicitly describe updates of a finite subset of the variables only. These variables will be called *system variables*. Thus, a run consists of infinitely many variables and we always graphically outline only the finite part of it concerning updates of the system variables, called the *restriction* of the run to the system variables. We assume that the values of all other variables change arbitrarily. Therefore, the set of all runs of a system will always be infinite. This set will be called the *behavior* of the system.

The next section provides the foundations of our formalism, in which we are able to specify such behaviors. The specification of the sample system M will be presented in Section 4.

3. The Logic TLDA

In this section we introduce the representation of the semantic model of TLDA, followed by its syntax and semantics.

3.1. The Semantic Model

The semantic model of a TLDA formula is a *run* as already intuitively introduced and exemplified above in (3)–(5). The notation of a run resembles that of an occurrence net known from Petri net's theory (see [7] for instance). A run consists of a *history* of each variable and a set of *transitions*.

History In a run σ of a system each variable $x \in Var$ evolves its *history*. A history of x is a finite or infinite sequence $H(x) = x_0\ x_1\ x_2\ \ldots$ of values $x_i \in Val$. x_i is the *local state* of x *at index* i. We abbreviate $H(x)$ to H_x. $l(H_x)$ denotes the length of a sequence H_x. As an example, $H_z = 0\,2\,4\ldots$ is the history of the variable z, and $H_z(0) = 0$, $H_z(1) = 2$ etc. The histories of the variables constitute the history of the run:

Let Val^+ and Val^ω denote the set of all non-empty, finite and infinite sequences of values, respectively, and let $Val^\infty \triangleq Val^+ \cup Val^\omega$. Then

$$H : Var \to Val^\infty$$

is a *history*.

Transitions Updates of different variables in a history may *synchronize*, i.e. occur coincidently. A transition is an update of one variable or a synchronized update of several variables; technically a mapping:

$$t : V \to \mathbb{N}_0$$

where $\emptyset \neq V \subseteq Var$ is finite. $V = dom(t)$ includes all variables that are *involved* in the transition t. $t(x) = i$ denotes that the ith value in the history of x is updated by t.

In the run σ_1 in (3) the transition t_1 depicts a synchronized update of the variables x and y. Hence x and y are involved in t_1. The transition t_{11} of the run σ_3 in (5) is an update of the variable y. Thus, y is the only variable involved in t_{11}.

Transitions in a run are (partially) ordered. We define an *immediate successor* relation $\prec\cdot$ for $t, u \in T$ by

$$t \prec\cdot u \quad \textit{iff there exists a variable } x \in dom(t) \cap dom(u) \textit{ with: } t(x) = u(x) - 1.$$

For example, $t_1 \prec\cdot t_2 \prec\cdot t_3 \prec\cdot \ldots$ holds in σ_1, $t_{11} \prec\cdot t_{22}$ does *not* hold in σ_3. Let $\prec$ denote the transitive closure of $\prec\cdot$.

Transitions in a run which are not related by $\prec$ are called *concurrent*. For instance, t_{11} and t_{22} are concurrent in σ_3.

We require that for every transition t there is a finite number of transitions t_i with $t_i \prec t$. This completes the notions required for the definition of a run:

Runs Let H be a history and let T be a set of transitions. $\sigma = (H, T)$ is a run iff

- For every variable $x \in Var$ and for all i with $0 \leq i < l(H_x) - 1$, there exists exactly one transition $t \in T$ with $t(x) = i$.
- For all $t \in T$ and for all variables $x \in dom(t)$ holds: $0 \leq t(x) < l(H_x) - 1$.
- The relation $\prec$ on T is irreflexive.

The runs σ_1–σ_3 fulfill these properties. In the rest of this section we assume a run $\sigma = (H, T)$ with a history H and a set of transitions T.

Since in a run σ the relation $\prec$ on T is transitive (by definition of a transitive closure) and irreflexive (by definition of the run), $\prec$ constitutes a *partial order* on the transition set of σ.

Cuts and Steps $\prec$ can canonically be generalized to the local states of variables. A set of local states which are not related by $\prec$ forms a *cut*.

Formally, a mapping

$$C : Var \to \mathbb{N}_0$$

is called a *cut* in σ iff for each $t \in T$ and all $x, y \in dom(t)$ holds:

$$\text{if } t(x) < C(x), \text{ then } t(y) < C(y).$$

For instance, C_0 with $C_0(x) = C_0(y) = C_0(z) = 0$, as well as C_2 with $C_2(y) = 2$ and $C_2(x) = C_2(z) = 1$ are cuts in σ_1.

$C_0 : Var \to \{0\}$ is obviously a cut in every run σ and will be called the *initial cut* in σ.

We say that a transition $t \in T$ *occurs at* C if t updates variables at the local states belonging to C, i.e. if $t(x) = C(x)$ for each $x \in dom(t)$.

From the definition of a cut arises an important observation: *Any two transitions that occur at C are concurrent in* σ.

When one or more transitions occur, another cut will be reached. For example, from C_0 in σ_1 the cut C_1 is reached by the occurrence of t_1.

Let U_C be the set of transitions that occur at C. For each cut C, the *successor cut* C' of C will be reached by occurrence of all transitions from U_C. C' is for each $x \in Var$ defined by:

$$C'(x) \triangleq \begin{cases} C(x)+1, & \text{if } x \in dom(t) \text{ for some } t \in U_C \\ C(x), & \text{otherwise.} \end{cases}$$

It is quite easy to prove that C' is a cut in σ, too.

A cut C and its successor cut C', together with the transition set U_C form a *step* S_C. Thus, S_C will canonically be defined by the cut C. Note that not every cut of a run σ can be reached by taking such *maximal* steps from C_0. Hence, the cuts do not constitute the run; they rather may be conceived as *observations* of the run.

3.2. Syntax of TLDA

Vocabulary A vocabulary of TLDA is given by the following sets: a set of function symbols $\mathscr{F}$, a set of predicate symbols $\mathscr{P}$ (the symbol for equality $=$ is one of them in particular), a set of special symbols and a set of variables.

Additionally, TLDA expressions can include brackets, which we use in order to overwrite the binding priorities or just to increase readability.

Each predicate symbol and each function symbol has an arity. Constants can be thought of as 0-arity functions.

The set of special symbols consists of the standard boolean connectives $\neg$ and $\wedge$, the quantifier $\exists$ and the temporal operator $\Box$. The sets $\mathscr{P}$, $\mathscr{F}$ and the set of special symbols should be pairwise disjoint.

An infinite set of variables Var_{all} is partitioned into infinite disjoint sets of:

- rigid variables $Var_{rigid} = \{\mathsf{m}, \mathsf{n}, \dots\}$,
- flexible variables $Var = \{x, y, \dots\}$,
- primed flexible variables $Var' \triangleq \{x' | x \in Var\} = \{x', y', \dots\}$,
- and $\sim$-variables $\widetilde{Var} \triangleq \{\widetilde{a} \mid \emptyset \neq a \subseteq Var\} = \{\widetilde{\{x\}}, \widetilde{\{y\}}, \widetilde{\{x,y\}}, \dots\}$.

$\widetilde{\{x\}}$ and $\widetilde{\{x,y\}}$ are abbreviated to $\widetilde{x}$ and $\widetilde{xy}$, respectively.

Terms The terms of our logic, like in the classical predicate logic, are made up of variables and functions applied to them:

- Any variable from $Var_{rigid} \cup Var \cup Var'$ is a term.
- If $t_1, t_2, \dots, t_n$ are terms, $f \in \mathscr{F}$ has arity n, then $f(t_1, t_2, \dots, t_n)$ is a term.

Formulas Based on the terms we can continue to define the formulas of our logic in the common way. The formulas are divided into two classes: the *step formulas* and the *run formulas*.

The set of step formulas over $\mathscr{F}$ and $\mathscr{P}$ is inductively defined as follows:

- If P is a predicate taking n arguments, $n \geq 1$, and if $t_1, t_2, \dots, t_n$ are terms over $\mathscr{F}$, then $P(t_1, t_2, \dots, t_n)$ is a step formula.
- Any variable from the set $\widetilde{Var}$ is a step formula.
- If F and G are step formulas, then so are $\neg F$ and $F \wedge G$.
- If $x \in Var_{rigid}$ and if F is a step formula, then so is $\exists x F$.

(We omit here the quantification over flexible and $\sim$-variables, which is also defined in TLDA, since this would extend the scope of this paper.)

The set of run formulas over $\mathscr{F}$ and $\mathscr{P}$ is inductively defined as follows:

- Any step formula F is a run formula.
- If F and G are run formulas, then so are $\neg F$, $F \wedge G$ and $\Box F$.
- If $x \in Var_{rigid}$ and if F is a run formula, then so is $\exists x F$.

We use some conventional arithmetical and logical abbreviations in TLDA, including boolean abbreviations *true* (for $P \vee \neg P$), *false*, $\Rightarrow$ and $\Leftrightarrow$, as well as $\Diamond$ (for $\neg\Box\neg$).

$x' \leq x + 1$, $\neg\widetilde{z} \wedge (\mathsf{m} \geq 0) \Rightarrow (z = 15)$, $\exists \mathsf{m}\,(\mathsf{m} = x^2)$, where m is a rigid variable, are examples of step formulas in our logic. $\Box(z < \mathsf{m})$, $\Box\widetilde{xy}$, $\Box(\neg\widetilde{z} \Rightarrow (x' = x + 5))$ are examples of run formulas, m is a rigid variable.

3.3. Semantics of TLDA

Now we explain briefly the difference between the sets of variables introduced above.

Rigid variables stand for an unknown but fixed value. Flexible variables will be mostly called *program* variables. They are intended to describe changes in our systems: Every program variable has a value in a particular cut C of a system run. A value of any given program variable in the successor cut C' will be described by a corresponding primed program variable.

The partition of variables into rigid and flexible variables is a well known idea (see for example [14]) and primed variables have also been used before for describing values of variables in a successor state ([12, 14]).

The $\sim$-variables are new. They are independent from the values assigned to the program variables in a cut of a system run: The $\sim$-variables can only take boolean values and provide information about the synchronization of variable updates. Some subsequent examples will clarify this concept.

The semantics of the logic resembles those for other temporal logics. We assume a non-empty set *Val* of concrete values, called the *universe*, we interpret each function symbol in $\mathscr{F}$ as a concrete function on *Val*, and each predicate symbol in $\mathscr{P}$ as a predicate over *Val*. Formally, the *interpretation* I of $(\mathscr{F}, \mathscr{P})$ consists of the following set of data:

- a non-empty set *Val*,
- for each n-ary $f \in \mathscr{F}$ a function $f^I : Val^n \to Val$, and
- for each $P \subseteq \mathscr{P}$ with n arguments a subset $P^I \subseteq Val^n$.

Evaluating terms Let r be a mapping $r : Var_{rigid} \to Val$ which associates with every rigid variable m a value $r(m)$ of the universe. The values of all program variables and primed program variables depend on a run.

Terms will be evaluated in steps of a run as follows: Let $\sigma = (H,T)$ be a run and let S_C be a step of σ taken from the cut C. To each rigid variable we assign its value according to the mapping r. To each program variable $x \in Var$ we assign the value $H_x(C(x))$, i.e. the value assigned to x at the $C(x)$th index in its history H_x. This is intuitively the value of x in the global state C. To each primed program variable $x' \in Var'$ we assign the value $H_x(C'(x))$. Intuitively, each variable x' gets the value of x in the succeeding global state C'.

Formally, to compute the value of a term in S_C under the interpretation I and with respect to r we inductively define a mapping r_C as follows:

$$\begin{aligned} r_C(m) &= r(m), && \text{if } m \in Var_{rigid},\\ r_C(x) &= H_x(C(x)), && \text{if } x \in Var,\\ r_C(x') &= H_x(C'(x)), && \text{if } x' \in Var', \text{ and}\\ r_C(f(t_1,\ldots,t_n)) &= f^I(r_C(t_1),\ldots,r_C(t_n)), && \\ & \quad \text{if } t_1,\ldots,t_n \text{ are terms.} \end{aligned}$$

Evaluating step formulas A model of a step formula consists of a step S_C of a run σ and an interpretation I of $(\mathscr{F}, \mathscr{P})$ (for convenience, we will write simply S_C in place of (S_C, I) when a model of a step formula is concerned). Let r be a valuation mapping of rigid variables.

We define the notion $S_C \,||\!=_r \psi$ of ψ holding in S_C with respect to r for each step formula ψ by structural induction on ψ:

- $S_C \,||\!=_r P(t_1,\ldots,t_n)$ iff $(r_C(t_1),\ldots,r_C(t_n)) \in P^I$.
- $S_C \,||\!=_r \widetilde{a}$ iff $a \subseteq dom(t)$ for any $t \in U_C$, i.e. we replace $\widetilde{a}$ by the boolean value *true*, if a is a subset of the variables involved in a transition t occurring at C, and by *false* otherwise.
- $S_C \,||\!=_r \neg F$, $S_C \,||\!=_r F \wedge G$ and $S_C \,||\!=_r \exists x F$ for $x \in Var_{rigid}$ are standard.

Examples Let S_{C_0} and S_{C_1} be steps of σ_1 as given in (3). In S_{C_0} holds $\neg\widetilde{z}$ because no transition of S_{C_0} involves z. In S_{C_1} holds $\widetilde{yz}$, since there is a transition t_2 in which both y and z are involved. This implies in particular that z is involved in this transition, so $\widetilde{z}$ is *true* in S_{C_1} too. The same holds for the variable y.

In contrast, in the initial step of the run σ_3 in (5) holds $\widetilde{y}$ and $\widetilde{z}$, because there are apparently transitions t_{11} and t_{22}, in which y and z are involved, respectively. But in this step $\widetilde{yz}$ does *not* hold.

Evaluating run formulas Now we extend the semantics to run formulas. A model of a run formula is a pair (σ, C) consisting of a run σ and a cut C of σ, and an interpretation I for $(\mathscr{F}, \mathscr{P})$ (we will write for convenience simply (σ, C) instead of (σ, C, I)). Analogously to step formulas, we define now the notion $(\sigma, C) \models_r \psi$ of ψ holding in (σ, C) with respect to a valuation mapping r of rigid variables for each run formula ψ by structural induction on ψ:

- $(\sigma, C) \models_r F$ iff $S_C \,||\!=_r F$.
- $(\sigma, C) \models_r \neg F$, as well as $F \wedge G$ and $\exists x F$ for $x \in Var_{rigid}$ are standard.
- $(\sigma, C) \models_r \Box F$ iff $(\sigma, C^*) \models F$ for every cut C^* of σ with $C^*(x) \geq C(x)$ for all $x \in Var$.

Notations We usually omit an explicit denotation of the mapping r and write simply $(\sigma, C) \models \Phi$ for $(\sigma, C) \models_r \Phi$. Furthermore, if a run formula Φ holds in σ at the initial cut C_0, i.e. $(\sigma, C_0) \models \Phi$, we write $\sigma \models \Phi$. The set of all models of Φ will be denoted by $\Sigma(\Phi)$. Hence, $\Sigma(\Phi)$ is the behavior of the system specified by Φ. Finally,

$$V(\Phi) \triangleq \{x \in Var \mid x, x' \text{ occurs in } \Phi \text{ or } \widetilde{a} \text{ occurs in } \Phi \text{ and } x \in a\}$$

denotes the set of *Var*-variables occurring in Φ.

4. Specifying Systems in TLDA

With the logic of Section 3 we are now ready to specify systems. In this section we revise our motivating example and describe the behavior of the system M based on the informal description from Section 2. We describe the initial values of the system variables x, y and z of M by the formula:

$$M_{init} \triangleq x = 1 \wedge y = 1 \wedge z = 0$$

Recall that there are two actions in the system M which are performed nondeterministically: The action A_1 swaps the values of x and y, the action A_2 reads y and changes z according to the current value of y. Each occurrence of A_1 or A_2 will be represented in a run of M by a transition involving

both x and y or y and z, respectively. We describe an update of a variable in case this variable is involved in a transition: x is involved only together with y, and the value of x will be set on the previous value of y. This will be expressed by the formula $\widetilde{x} \Rightarrow (\widetilde{xy} \wedge x' = y)$. We likewise describe what happens if y and z are involved in a transition. Additionally, we claim that x and z are never involved in the same transition. Hence, the following formula specifies the updates of M:

$$\begin{aligned} M_{next} \triangleq \Box\, ((\widetilde{x} \Rightarrow (\widetilde{xy} \wedge x' = y)) \wedge \\ (\widetilde{y} \Rightarrow (\widetilde{xy} \wedge y' = x \vee \widetilde{yz} \wedge y' = y)) \wedge \\ (\widetilde{z} \Rightarrow (\widetilde{yz} \wedge z' = z + y + 1)) \wedge \\ \neg\widetilde{xz}) \end{aligned}$$

We focused so far on the *safety* part of the specification of M only. Now we consider the *liveness* condition for M, stating that each of the actions A_1 and A_2 should be executed infinitely often. In order to satisfy this condition it suffices to require that both x and z are infinitely often involved in a transition of a run (such requirement for y would be redundant):

$$L \triangleq \Box \Diamond \widetilde{x} \wedge \Box \Diamond \widetilde{z}$$

For simplicity we omit here the general definition of fairness. Hence, the system M is specified by $M_{init} \wedge M_{next} \wedge L$.

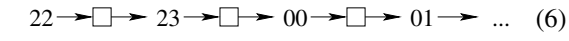
Note that the runs σ_1 and σ_2 can now be well distinguished: The run σ_1 is a model of this specification, while σ_2 is not, since $\Box \Diamond \widetilde{x}$ does not hold in σ_2. Consequently, σ_2 is excluded from the behavior of M.

5. Composing Specifications

In this section, we focus on *parallel composition* of systems and their specification in TLDA. Let S_1 and S_2 be systems specified by formulas Φ_1 and Φ_2, respectively. Parallel composition of S_1 and S_2 is defined as the intersection of their behaviors $\Sigma(\Phi_1) \cap \Sigma(\Phi_2)$. From this definition follows immediately by logical reasoning that the specification formula of the composed system is the *conjunction* $\Phi_1 \wedge \Phi_2$ of the specification formulas of the components. The idea of composition as conjunction has been suggested in [4, 1, 2, 13]. Works on compositional semantics based on partial order are [10, 11].

We introduce the basic concepts with a simple version of a clock composition. We borrow this example from [13].

An *hour clock* displays the hours; for this purpose we assume the variable *hr* to display sequences such as 22 23 00 01 .... Likewise, a minute clock with the variable *min* displays sequences such as 58 59 00 01 ....



(6)

is a run of the hour clock. Recall that this run actually consists of infinitely many variables. All variables other than *hr*, including *min*, may change arbitrarily. These variables constitute the *environment* of *hr*. A run of the minute clock resembles (6) with some obvious modifications.

We start with specifying the hour clock: The formula HR_{init} specifies the clock's initial state, viz the initial value of the variable *hr* to vary between 0 and 23. HR_{next} specifies the clock's updates: Each update increases *hr* by one, with the exception that 23 is followed by 0.

$$\begin{aligned} HR_{init} &\triangleq hr \in \{0, \ldots, 23\} \\ HR_{next} &\triangleq hr' = suc_{hr}(hr) \\ &\quad \text{with } suc_{hr}(23) = 0,\ suc_{hr}(n) = n + 1, \text{ if } n \neq 23 \end{aligned}$$

Since the hour clock is a detached component which should later work as a part of a bigger system, we specify it in a way allowing arbitrary synchronization with any other subsystem. Such specification will be called *environment invariant*.

Formally, a formula Φ will be called environment invariant iff for all runs σ with $\sigma \models \Phi$ holds: $\sigma* \models \Phi$ for all runs $\sigma*$ such that the restrictions of σ and of $\sigma*$ to the variables $V(\Phi)$ are identical.

We give here a sufficient *syntactical* condition for environment invariance: A formula Φ is environment invariant if either $\Phi \equiv \mathit{false}$, or one of the cases 1–3 holds:

1. no primed or $\sim$-variables occur in Φ.
2. Φ has the form $\widetilde{a} \Rightarrow \Psi$ where Ψ is a step formula such that either
   - $\Psi \equiv \mathit{false}$ (i.e. Φ is equivalent to $\neg\widetilde{a}$), or
   - if v or v' occurs in Ψ then $v \in a$ and if $\widetilde{v}$ occurs in Ψ then $v \cap a \neq \emptyset$.
3. Φ has the form $\Psi \Rightarrow \widetilde{a}$ where Ψ is a step formula such that
   - no $\sim$-variables occur in Ψ, and
   - $V(\Psi) = \{v\}$ for a variable $v \in \mathit{Var}$ and $v \in a$.

Lemma 1 *Let Φ and Ψ be formulas. If Φ and Π are environment invariant, then so are $\Phi \wedge \Pi$ and $\Box\Phi$.*

These properties are very useful for writing system specifications.

Observe that the formula M_{next} in Section 4 is environment invariant while $\Box\,(\widetilde{x} \wedge x' = x + 1)$ and $\Box\,(\widetilde{xy} \wedge x' = y)$ are not.

Now, we come back to the clock example. To allow *hr* an arbitrary synchronization with its environment, we require that the formula HR_{next} is to be applied only in system steps which involve the variable *hr*. Observe that this fulfills the syntactical condition given above. Hence, the specification of the hour clock is

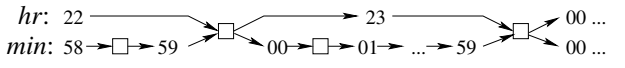
$$HR \triangleq HR_{init} \wedge \Box\,(\widetilde{hr} \Rightarrow HR_{next})$$

The specification of the minute clock strongly resembles the

specification of the hour clock:

$$\begin{aligned} MIN_{init} &\triangleq min \in \{0,\ldots,59\} \\ MIN_{next} &\triangleq min' = suc_{min}(min) \\ &\quad \text{with } suc_{min}(59) = 0,\ suc_{min}(n) = n+1, \text{ if } n \neq 59 \\ MIN &\triangleq MIN_{init} \wedge \Box\,(\widetilde{min} \Rightarrow MIN_{next}) \end{aligned}$$

One easily observes that every run of the hour-minute clock, such as (7), is a model of the conjunction $HR \wedge MIN$. This conjunction, however, specifies the hour and the minute clock working really in *parallel*. Thus, it additionally admits models that are not proper runs of the hour-minute clock.



(7)

The unwanted models do not properly *synchronize* the updates of *hr* and *min*. Hence we strive for an additional formula, *SYNC*, to express additional constraints on the models. *SYNC* talks about synchronized updates of *hr* and *min* by help of the variable $\widetilde{hrmin}$. Two properties are required: Firstly, the variables *hr* and *min* synchronize their updates iff $min = 59$. Secondly, each update of *hr* is synchronized with an update of *min*:

$$SYNC \triangleq \Box(\widetilde{hrmin} \Leftrightarrow min = 59) \wedge \Box(\widetilde{hr} \Rightarrow \widetilde{hrmin})$$

Note that the formula *SYNC* is environment invariant, too. (But this is not always necessary.) It can easily be shown by transforming $\widetilde{hrmin} \Leftrightarrow min = 59$ into an equivalent formula $(\widetilde{hrmin} \Rightarrow min = 59) \wedge (min = 59 \Rightarrow \widetilde{hrmin})$ fulfilling the syntactical condition given above, and then by applying Lemma 1.

Hence, the hour-minute clock will be specified by

$$HR \wedge MIN \wedge SYNC.$$

Since this specification is by Lemma 1 also environment invariant, the hour-minute clock can effortless be used as a component for a further system.

The above clock example shows how a system composed of independent interacting components will be usually specified: Firstly, an environment invariant specification will be given for each component. Since the components are independent they always have disjoint system variables. Secondly, a synchronization formula employing primarily $\sim$-variables will be added to define the interactions between the components.

6. Conclusion

We suggest a new temporal logic, TLDA, for specifying and verifying distributed systems. The logic can syntactically be conceived as a variant of TLA. TLDA, however, is interpreted on partial order semantics. This renders the logic more expressive. Furthermore, we have shown that TLDA supports a compositional system design: subsystems can be specified separately and then be integrated into one system.

References

[1] M. Abadi and L. Lamport. Decomposing specifications of concurrent systems. In E.-R. Olderog, editor, *Proc. of the Working Conference on Programming Concepts, Methods and Calculi (PROCOMET '94)*, volume A-56 of *IFIP Transactions*, pages 327–340. North-Holland, 1994.

[2] M. Abadi and L. Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–534, May 1995.

[3] M. Abadi and S. Merz. On TLA as a logic. In M.Broy, editor, *Deductive Program Design*, NATO ASI series F. Springer-Verlag, 1996.

[4] M. Abadi and G. Plotkin. A logical view of composition. *Theoretical Computer Science*, 114(1):3–30, 1993.

[5] A. Alexander and W. Reisig. Logic of involved variables - system specification with Temporal Logic of Distributed Actions. In *Proc. of the 3rd International Conference on Aplication of Concurrency to System Design (ACSD'03)*, pages 167–176, Guimaraes, Portugal, 2003.

[6] H. Barringer, R. Kuiper, and A. Pnueli. Now you may compose temporal logic specifications. In *Proc. of the 16th Annual ACM Aymposium on Theory of Computing*, pages 51–63, 1984.

[7] E. Best and C. Fernandez. Nonsequential processes – a Petri net view. In W. Brauer, G. Rozenberg, and A. Salomaa, editors, *EATCS Monographs on Theoretical Computer Science*, volume 13. Springer-Verlag, 1988.

[8] A. Cau and W.-P. d. Roever. A dense-time temporal logic with nice compositionality properties. In *Proc. of the 6th International Workshop on Computer Aided Systems Theory EUROCAST'97, Las Palmas de Gran Canaria, Spain*, volume 1331 of *LNCS*, pages 123–145. Springer, February 1997.

[9] J. Fiadeiro and T. Maibaum. Sometimes "tomorrow" is "sometime": Action refinement in a temporal logic of objects. In D. Gabbay and H. Ohlbach, editors, *Proc. of the 1st International Conference on Temporal Logic ICTL'94*, volume 827 of *LNAI*, pages 48–66. Springer-Verlag, 1994.

[10] D. Gomm, E. Kindler, B. Paech, and R. Walter. Compositional liveness properties of EN-systems. In M. Marsan, editor, *Applications and Theory of Petri Nets 1993, 14th International Conference*, volume 691 of *LNCS*, pages 262–281. Springer-Verlag, June 1993.

[11] E. Kindler. A compositional partial order semantics for Petri net components. In P. Azéma and G. Balbo, editors, *Application and Theory of Petri Nets 1997, 18th International Conference*, volume 1248 of *LNCS*, pages 235–252. Springer-Verlag, June 1997.

[12] L. Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[13] L. Lamport. Composition: A way to make proofs harder. In A. W.P.de Roever, H.Langmaack, editor, *Compositionality: The Significant Difference, International Symposium, COMPOS'97*, volume 1536 of *LNCS*, pages 402–423, September 1997.

[14] Z. Manna and A. Pnueli. *The temporal logic of Reactive and Concurrent Systems: Specification.* Springer, 1992.

[15] A. Mokkedem and D. Mery. A stuttering closed temporal logic for modular reasoning about concurrent programs. In D. Gabbay and H. Ohlbach, editors, *Temporal Logic, Proc. of the 1st International Conference on Temporal Logic ICTL'94*, volume 827 of *LNAI*, pages 382–397. Springer-Verlag, 1994.

[16] A. Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13(1):45–61, 1981.

[17] M. Reynolds. Changing nothing is sometimes doing something: Fairness in extensonal semantics. ulr: citeseer.nj.nec.com/ reynolds96changing. html, 1996.