Noname manuscript No.
(will be inserted by the editor)

A Formal Model of Composing Components

The TLA$^+$ Approach

Ondrej Rysavy · Jaroslav Rab

Received: date / Accepted: date

Abstract In this paper, a method for writing composable TLA$^+$ specifications that conform to the formal model called Masaccio is introduced. Specifications are organized in TLA$^+$ modules that correspond to Masaccio components by means of a trace-based semantics. Hierarchical TLA$^+$ specifications are built from atomic component specifications by parallel and serial composition that can be arbitrary nested. While the rule of parallel composition is a variation of the classical joint-action composition, the authors do not know about a reuse method for the TLA$^+$ that systematically employs the presented kind of a serial composition. By combining these two composition rules and assuming only the noninterleaving synchronous mode of an execution, the concurrent, sequential, and timed compositionality is achieved.

Keywords Composing Specifications · Component Model · Hierarchical Specifications · Synchronous Mode of Executions · Temporal Logic of Actions

1 Introduction

Software running in embedded systems necessary acquires some properties of the physical world. Usually, these properties form a part of non-functional aspects in the software requirements [4]. Among them, the timeliness is the most important one for the class of real-time embedded software. To model embedded software, non-functional aspects must be considered by a specification method otherwise the model easily diverges from the reality and becomes inapplicable in further refinement and analysis. Constructing large computer-based systems relies on the effective and systematic application of a modular approach. Various entities playing the role of composable building blocks have been proposed, most notably, classes for object-oriented program construction, components in hardware design, procedures and modules in procedural programming, and active objects and actors for reactive programming [18].

This paper deals with a method based on a formalism called Temporal Logic of Actions [16] that is capable to describe embedded control software in a modular manner and apply an automatized model-checker tool to verify required functional and certain non-functional properties of a specification. The main contribution of this paper lies in the demonstration of how to systematically develop TLA$^+$ specifications whose interpretation is that of a formal model called Masaccio[7]. This formal model allows for the definition of a component hierarchy that is built from atomic components using operations of parallel composition, serial composition, renaming of variables, renaming of locations, hiding of variables, and hiding of locations. As the resulting TLA$^+$ specifications have the form of a conjunction of initial predicate and next-state actions, they are readily explorable by the TLA$^+$ explicit model-checker.

Ondrej Rysavy · Jaroslav Rab
FIT UIFS, Brno University of Technology
Bozetechova 2, 61266 Brno, Czech Rep.
Tel.: +420-54114-1118
Fax: +420-54114-1270
E-mail: rysavy@fit.vutbr.cz
E-mail: rabj@fit.vutbr.cz

1.1 Related Work

The presented approach stems from the research on a compositional semantics that have been studied, for instance, for hybrid automata [19], algebraic specifications [5], Hoare logic [11], synchronous languages [20],

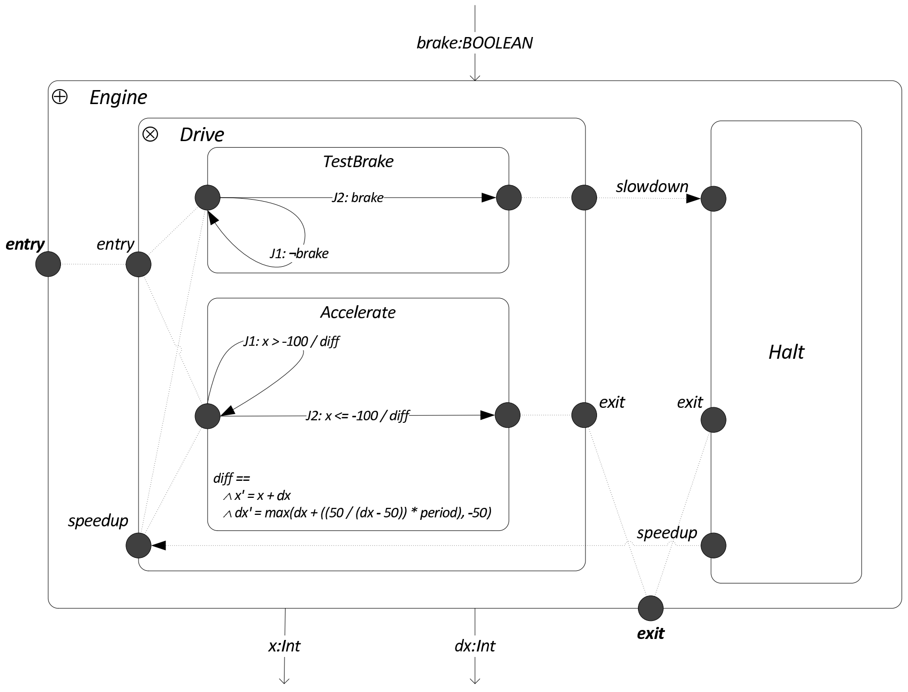

Fig. 1 The component Engine

process algebra [14], and in the frame of architecture description languages, e.g. CODE, Modechart, Wright, MetaH, and AADL. In particular, we refer to Masaccio component model [7] that allows for arbitrary nested serial and parallel composition and application of assume-guarantee principle [9] for the purposes of hierarchical system design, development and analysis. The systematic approach to composing TLA$^+$ specifications can be found in [2] and [3], which assumes the decomposition of a solid system specification into a conjunctively combined parts. Hermann et.el. defined an extension of TLA$^+$ with explicit notion of process [10]. The presented work is based on the principles initially elaborated in [21].

2 The Component Model

This section gives a brief overview of a formal model for embedded components as defined by Henzinger in [7]. In this paper, only discrete components are considered, although the proposed approach employs a specification language that can be used for description of hybrid systems as well [15].

A fundamental entity of the model is a component. A component A consists of the definition of an interface and an internal behavior. An interface defines disjoint sets of input variables, V_A^{in}, output variables, V_A^{out}, and a set of public locations, L_A^{intf}. An execution of the component consists of a finite sequence of jumps. A jump is a pair $(p, q) \in [V_A^{in,out}] \times [V_A^{in,out}]$[1]. An observation p is called the source of jump (p, q) and an observation q is called the sink of jump (p, q). We write $p[v]$ for the value of a variable $v \in V_A^{in,out}$ in an observation p. A jump v is successive to a jump u if the source of jump v is equal to the sink of jump u. Formally, an execution of A is a pair (a, w) or a triple (a, w, b), where $a, b \in L_A^{intf}$ are interface locations and $w = w_0 \dots w_n$ is a nonempty, finite sequence of jumps of A such that every jump w_i, for $1 \le i \le n$ is successive to the immediately preceding step w_{i-1}. We write E_A for the set of executions of the component A.

An atomic component is the basic form of components found in Masaccio. The behavior of an atomic component is solely specified by its jump action. A jump action J is given by a predicate φ_J^{jump} defined on variables in $X_J \cup Y'_J \cup Z'_J$, where

- X_J is a finite set of source variables,
- Y_J is a finite set of uncontrolled sink variables, and
- Z_J is a finite set (disjoint to Y_J) of controlled sink variables.

For instance, the jump action predicate

$$\varphi_J^{jump} \triangleq x \le -100 \land x' = x + dx \land dx' \\ = max(dx + ((50/(dx - 50) * period), -50)$$

contains source variables x and dx those are also controlled sink variables. The variable *period* is also a source

[1] $[V_A^{in,out}]$ stands for a set of all possible assigments of values into input and output variables of a component.

variable, but it is not a sink variable. The interface of an atomic component publish input variables read by the component and output variables controlled by the component, which is defined as follows:

– input variables $V^{in}_{A(J)} = (X_J \setminus Z_J) \cup Y_J$, and
– output variables $V^{out}_{A(J)} = Z_J$.

The component $A(J)$ has two interface locations, *from* and *to*; that is, $L^{intf}_{A(J)} = \{\mathit{from}, \mathit{to}\}$. The entry condition of *from* is the projection of the jump predicate to the source variables and the primed uncontrolled sink variables, that is $\varphi^{en}_{A(J)}(\mathit{from}) = (\exists Z'_J \varphi^{jump}_J)$. The entry condition of *to* is unsatisfiable.

Two components A and B can be combined to form a *parallel composition* $C = A \otimes B$ if the output variables of A and B are disjoint and for each interface location a common to both A and B, the entry conditions of a are equivalent in A and in B. The input variables of the component C form a set $V^{in}_C = (V^{in}_A \setminus V^{out}_B) \cup (V^{in}_B \setminus V^{out}_A)$. The output variables of the component C form a set $V^{out}_C = V^{out}_A \cup V^{out}_B$. The interface locations of $A \otimes B$ are the interface locations of A together with the interface locations of B. An interface location a, which is common to A and B and its entry conditions agree in both components, has this entry condition also in $A \otimes B$. Other interface locations cannot be used to entry the component.

The set of executions of a component $C = A \otimes B$ is defined as 1) the pair (a, w) is an execution of $A \otimes B$ iff $(a, w|_A)$ is an exection of A and $(a, w|_B)$ is an execution of B[2], 2) the triple (a, w, b) is an execution of $A \otimes B$ iff either $(a, w|_A, b)$ is an execution of A and $(a, w|_B)$ is an execution of B, or $(a, w|_B, b)$ is an execution of B and $(a, w|_A)$ is an execution of A.

The definition of parallel composition specifies that each jump of both subcomponents is performed in synchronous manner. Moreover, if one component reaches an exit interface location then the execution of the other component is terminated. If both components reach their exit locations one is chosen nondeterministically. As the consequence of these properties, parallel composition operation is associative and commutative.

Two components A and B can be composed in series to form a *serial composition* $C = A \oplus B$ if the set of output variables are identical; that is, $V^{out}_A = V^{out}_B$. The input variables of the composed component is $V^{in} = V^{in}_A \cup V^{in}_B$. The interface locations of $A \oplus B$ are the interface locations of A together with the interface locations of B. If a is an interface location of both A and B, then the entry condition of a in $A \oplus B$ is the disjunction of the entry conditions of a in the subcomponents A and B.

[2] Expression $w|_C$ denotes a restriction of the trace w to values for the I/O variables of the component C.

The set of executions of the component $C = A \oplus B$ contains 1) the pair (a, w) iff either $(a, w|_A)$ is an execution of A, or $(a, w|_B)$ is an execution of B, 2) the triple (a, w, b) iff either $(a, w|_A, b)$ is an execution of A, or $(a, w|_A, b)$ is an execution of B. The operator of serial composition is associative, commutative, and idempotent.

To support these two compositional operations, renaming and hiding operations are defined for variables and locations. The renaming operation maps variables and locations of different names to each other that allows for sharing data and control between components. Hiding makes variables or locations internal to the component, which is useful when a complex behavior is modeled inside the component.

3 TLA$^+$

The Temporal Logic of Actions (TLA) is a variant of the linear-time temporal logic. It was developed by Lamport [16] primarily for specifying distributed algorithms, but several works shown that the scope of applications is much broader. The system of TLA$^+$ extends TLA with data structures allowing for easier description of complex specification patterns. TLA+ specifications are organized into modules. Modules can contain declarations, definitions, and assertions by means of logical formulas. Declarations consist of constants and variables. Constants can be uninterpreted until an automated verification procedure is used to verify the properties of the specification. A collection of values assigned to variables describe completely a state of the system being analyzed. Transition formulas asserts the values of the variables in every two successive states. A specification of the system is given by means of a temporal formula defined as a conjunction of the form $\mathit{Init} \wedge \Box[\mathit{Next}]_v \wedge L$, where *Init* is the initial condition, *Next* is the next-state relation (composed from transition formulas), and L is a conjunction of fairness properties. Transition formulas, also called actions, are ordinary formulas of an untyped first-order logic defined on a denumerable set of variables, partitioned into sets of rigid variables, flexible and primed flexible variables. Transition formulas can reference these variables to express a relation between two consecutive states. The generation of a transition system for the purpose of model checking verification or simulation is governed by the enabled transition formulas. The formula $\Box[\mathit{Next}]_v$ admits transitions that leave a set of variables v unchanged. This is known as stuttering, which is a key concept that enables the refinement and compositional specifications. The initial condition and the next-state relation specify the possible behavior of the system. Fairness conditions strengthen the specification by asserting that given actions must occur. The TLA$^+$ does not formally distinguish between a system specification and a property specification. Both specifications are expressed as formulas of the temporal logic and are connected by the implication $S \Rightarrow F$, where S is a specification and F is a property. Confirming the validity of this implication stands for showing that the specification S has the property F. The TLA$^+$ is accompanied with a set of tools. The TLA+ model checker, called TLC, is a state-of-the-art model analyzer that attempts to explore all reachable states in finite TLA+ models. The input to TLC consists of a specification file and a configuration file, which gives an interpetation to constants, which is necessary for model computation procedure. An execution of the TLC produces a result that gives an answer to the question of whether $S \Rightarrow F$. In the case of negative answer, the execution violating the implementation is presented to the user. A technique known as symmetry reduction is implemented helping to scale the verification method for larger specifications.

4 Specification of Components

Using a simple example as required by space constraints, this section explains the construction of TLA$^+$ specifications that corresponds to Masaccio embedded components.

The example represents a specification of a component *Engine* taken from [7]. This component is a part of a more complex specification that models the control of a railway crossing. In particular, the *Engine* component controls acceleration and deceleration of a train that is moving in a near distance to the railway crossing. Although this example is rather trivial, it is sufficient to demonstrate basic principles of the specification method as it contains both parallel and serial compositions.

The component *Engine* and its subcomponents are modeled in figure 1. The components are represented by rectangles with rounded corners. Input and output variables are represented by open arrows connected to component boundaries. The position of an arrow determines the direction of a variable. The locations are drawn as solid disks and are positioned at the boundaries of components. Jump actions are represented by solid arrows labeled with condition predicates and action predicates.

The component *Engine* consists of a serial composition of two subcompoments, namely, *Drive* and *Halt*. An entry location is directly connected with one of the locations of *Drive* component. There is one exit location that is accessible from both subcomponents. Other interface locations, namely *slowdown* and *speedup*, serve for passing the control between *Drive* and *Halt* components. Further, the component interacts with the environment by reading an input variable *brake* and controlling output variables x and dx. These variables are also available to both subcomponents.

The component *Drive* governs train acceleration. The component is a parallel composition of two atomic components, namely *TestBrake* and *Accelerate*. The input variable *brake* determines whether the train accelerates or decelerates. Its value is observed by *TestBrake* component that removes the control from *Drive* component if the variable *brake* signalizes the application of emergency brake. In component *Accelerate*, the actual speed and the distance of the train to the rail crossing is computed.

The component *Halt* has similar structure to the component *Drive*. It's purpose is to slow the train down until *brake* is released. After that the control passes back to *Drive* component through the location *speedup*.

To show that TLA$^+$ specifications conform to Masaccio interpretation, the interpretation of TLA$^+$ expressions needs to be defined. The following simplified system is used (for complete semantics see e.g. [12]). The meaning of a module depends on a context. A basic context consist of declarations and definitions of all built-in operators of TLA$^+$ together with definitions and declarations of modules extended or instantiated. The meaning of a module $\mathcal{M}$ in a context $\mathcal{C}$ is given by the following sets:

- $Dcl^{\mathcal{C}}_{\mathcal{M}}$ is a set of declarations.
- $Def^{\mathcal{C}}_{\mathcal{M}}$ is a set of definitions.
- $Asm^{\mathcal{C}}_{\mathcal{M}}$ is a set of assumptions.
- $Thm^{\mathcal{C}}_{\mathcal{M}}$ is a set of theorems.

The TLA$^+$ module can examined by means of simulation and formal verification if it contains a definition in the standard form:

$$Init \wedge \Box[Next]_{vars} \wedge Temporal$$

where *Init* is the inital predicate, *Next* is the next-state action, *vars* is the tuple of all variables, and *Temporal* is a temporal formula that usually specifies a liveness condition. A set of behaviors can computed such that it satisfies the specification given by this definition. Using a model-checking technique the computational method tries to find a reachable graph describing all states and behaviors satisfying the specification; that is a smallest graph $\mathcal{G}^{spec}_{\mathcal{M}_{\mathcal{C}}} =$ consisting of

- $N^{spec}_{\mathcal{M}_{\mathcal{C}}}$, which is a set of all reachable states of module $\mathcal{M}$ in a context $\mathcal{C}$ with specification given by a

```
┌──────────────────────────── MODULE Accelerate ────────────────────────────┐
EXTENDS Reals, Components
VARIABLES x, dx  Output variable
├───────────────────────────────────────────────────────────────────────────┤
diff ≜
    ∧ x' = x + dx
    ∧ dx' = max(dx + ((50/(dx − 50)) ∗ period), − 50)

J1 ≜ JumpAtom(TRUE)
    ∧   x > − 100 ∧ diff

J2 ≜ JumpAtom(FALSE)
    ∧   x ≤ − 100 ∧ diff

├───────────────────────────────────────────────────────────────────────────┤
Init  ≜ InitAtom ∧ dx ∈ Real ∧ x ∈ Real
Next  ≜ J1 ∨ J2
└───────────────────────────────────────────────────────────────────────────┘
```

Fig. 2 The TLA$^+$ specification of Accelerate

standard form definition $spec \in Def_{\mathcal{M}}$. At least, the set contains all states s in which *Init* is satisfied.

- $E^{spec}_{\mathcal{M}_\mathcal{C}}$, which is a set of edges such that $(r, s) \in E^{spec}_{\mathcal{M}_\mathcal{C}}$ iff the unprimed fraction of *Next* is satisfied in state r and the primed fraction of *Next* is satisfied in state s.
- $D^{spec}_{\mathcal{M}_\mathcal{C}}(s, x)$, which is a value assigning function that for each variable $x \in Dcl_{\mathcal{M}}$ gives its actual value in the given state s.

We write $s \models_{\mathcal{G}^{spec}_{\mathcal{M}_\mathcal{C}}} F$ for asserting that a formula F is satisfied in a state s of the reachability graph $\mathcal{G}^{spec}_{\mathcal{M}_\mathcal{C}}$. Similarly, $(s, r) \models_{\mathcal{G}^{spec}_{\mathcal{M}_\mathcal{C}}} A$ asserts that action formula A is satisfied in a pair of states $(s, r) \in E^{spec}_{\mathcal{M}_\mathcal{C}}$. If $s \models_{\mathcal{G}^{spec}_{\mathcal{M}_\mathcal{C}}} F$ for all $s \in N^{spec}_{\mathcal{M}_\mathcal{C}}$ we simply write $\models_{\mathcal{G}^{spec}_{\mathcal{M}_\mathcal{C}}} F$.

Note that Masaccio interpretation is defined in terms of execution traces. Obviously, an execution trace corresponds very closely to TLA$^+$ behavior, which can be generated by traversing a graph $\mathcal{M}$. This similarity will be very useful in the further development, where rules of conformance of TLA$^+$ specifications and Masaccio models are built upon the execution semantics.

4.1 Specifying atomic components

According to Masaccio semantics, an atomic discrete component $A(J)$ is completely specified by a jump predicate that defines a set of legal jumps J. Further, an atomic component has an arbitrary number of input and output variables. In each atomic component, there are only two interface locations, denoted as *from* and *to*.

The representation of an atomic component is straightforward in TLA$^+$ language. In figure 2, TLA$^+$ description of *Accelerate* component is shown. A set of jumps is a conjunction of two next-state actions. Action $J1$ is executed if train is moving in a near distance. Action $J2$ is enabled if the train leaves the perimeter of the rail crossing. The differential computation is done in function *diff*. Train acceleration is expressed as $((50/(dx - 50)) * period)$, where *period* defines an amout of time that equals time difference dt. This constant comes, together with a bundle of other declaration and definition from module *Components* (see appendix for the full specification of this module) which is open in the module *Accelerate* using EXTENDS statement.

An atomic component $A(J)$ is equally represented by a TLA$^+$ module $\mathcal{M}_{A(J)} = \langle Dcl_{A(J)}, Def_{A(J)} \rangle$ with a reachability graph $\mathcal{G}_{A(J)} = \langle N_{A(J)}, E_{A(J)}, D_{A(J)} \rangle$ induced by the specification $Init \wedge \Box[Next]_{vars}$ if the following conditions are satisfied:

- Module M declares a variable for each I/O variable of the atomic component; that is, for every $v^T \in V^{in,out}_{A(J)}$ there is $v \in Decl_{A(J)}$ such that $\models_{\mathcal{G}_{A(J)}} v \in T$. Notation v^T states that set T is a domain of variable v [3].
- The meaning of next-state action *Next* agrees with the predicate $\varphi^{jump}_J(a)$; that is, for every jump $w = (p, q)$:
  - if (a, w) is an execution of $A(J)$ then there is a pair of states $(\hat{p}, \hat{q})$ such that $(\hat{p}, \hat{q}) \models_{\mathcal{G}_{A(J)}} N$ and location a is active in both states $\hat{p}$ and $\hat{q}$.
  - if (a, w, b) is an execution of $A(J)$ then there is a pair of states $(\hat{p}, \hat{q})$ such that $(\hat{p}, \hat{q}) \models_{\mathcal{G}_{A(J)}} N$. Location a is active only in state $\hat{p}$ and location b is active only in state $\hat{q}$.

  Moreover, it has to hold for both cases that values of every variable of $v \in V_{A(J)}$ in both interpretations are equal; that is, $D(\hat{p}, v) = p[v]$.

[3] In Masaccio, the notation $v : T$ is used but we stick to v^T to avoid confusion if overloading operator ':'.

```
1  ---------------------------- MODULE Drive ----------------------------
2  EXTENDS Components, Integers, Sequences
3  VARIABLES brake  The input variable.
4  VARIABLES x, dx  Controlled variables.
5  ----------------------------------------------------------------------
6  driveBrake ≜ INSTANCE TestBrake WITH current ← "DriveBrake"
7  accelerate  ≜ INSTANCE Accelerate WITH current ← "Accelerate"
8  ----------------------------------------------------------------------
9  Init ≜ driveBrake!Init ∧ accelerate!Init
10    ∧ InitParallel({"DriveBrake", "Accelerate"})

12 Next ≜ driveBrake!Next ∧ accelerate!Next
13    ∧ JumpParallel({"DriveBrake", "Accelerate"})

15 Activate ≜ driveBrake!Activate ∧ accelerate!Activate ∧ ActivateCurrent

17 Suspend ≜ SuspendCurrent ∧ driveBrake!Suspend ∧ accelerate!Suspend
18 ----------------------------------------------------------------------
19 L_slowdown ≜ activity["DriveBrake"] = FALSE
20 L_exit ≜ activity["Accelerate"] = FALSE
21 ======================================================================
```

Fig. 3 The TLA$^+$ specification of Drive

– The meaning of initial predicate *Init* agrees with the predicate $\varphi^{en}_{A(J)}(\mathit{from})$; that is, for every trace of atomic component $A(J)$ with prefix $(\mathit{from}, (p, q))$ there exists a state $\hat{p}$ that corresponds to state p and $\hat{p} \models_{\mathcal{G}_{A(J)}} \mathit{Init}$.

As it can be seen from the TLA$^+$ specification of *Accelerate* module in figure 2, the jump actions of atomic specification contains reference to *JumpAtom* function that depends on the variable *clock*. This variable serves to synchronization purposes. It enforces that parallel actions are executed at the same time. Therefore all jumps include the condition stating $\mathit{clock}' = \neg \mathit{clock}$. Except proper actions, there are also specific actions supporting serial compositions as described later in this section. These specific actions violate this condition requiring that the time is stopped; that is, $\mathit{clock}' = \mathit{clock}$.

4.2 Specifying Parallel Composition of Components

The component *Drive*, which is a part of component *Engine* shown in figure 1, is a result of parallel composition of two subcomponents. The corresponding TLA$^+$ specification is given in figure 3. The semantics of parallel composition corresponds to joint-action specification as described by Lamport in [16, p.147]. Its encoding in TLA is straightforward.

The *Drive* module contains input and output variables *brake*, *x*, *dx*. The module instantiates its submodules - *TestBrake* and *Accelerate*. The constant *current* of each submodule is set to corresponding component identification, which allows for referencing the subcomponents in the specification. Line 9 defines a collection of initial states of the subcomponents. The initial predicate *Init* is a conjunction of the initial predicates of all its subcomponents and *InitParallel* predicate that is defined in *Components* module. This state predicate gives an initial value to the element of *activity* array that corresponds to the composed component. In this case, it is *activity*[*current*], where the value of *current* constant is allowed to be given in a component that includes *Drive* as its subcomponent. In fact, such a component is the component *Engine* as shown later. For parallel composition, the value of *activity*[*current*] amounts to the value given by logical conjuction of activity values of all its immediate subcomponents. The state predicate *InitParallel* is defined as follows:

$$
\begin{aligned}
&\mathit{InitParallel}(S) \triangleq \\
&\quad \land\ \mathit{activity} \in [\mathit{Components} \rightarrow \text{BOOLEAN}\ ] \\
&\quad \land\ \mathit{activity}[\mathit{current}] = \forall\, s \in S : \mathit{activity}[s]
\end{aligned}
$$

The next-state action predicate *Next* is a conjunction of next-state predicates of subcomponents, which gives the intended execution interpretation of the component; that is, the jumps of subcomponents are executed in parallel and in synchronous manner. This style of specification is usually known as joint-action non-interleaving composition [1]. The next-state action predicate of the parallely composed component is supplied with *JumpParallel* predicate as the new value of *activity*[*current*] element has to be computed after every execution of the action. The predicate *JumpParallel* is defined in *Components* module as follows:

$$
\begin{aligned}
&\mathit{JumpParallel}(S) \triangleq \\
&\quad \land\ \mathit{activity}[\mathit{current}] = \text{TRUE} \\
&\quad \land\ \mathit{activity}' \in [\mathit{Components} \rightarrow \text{BOOLEAN}\ ] \\
&\quad \land\ \mathit{activity}'[\mathit{current}] = \forall\, s \in S : \mathit{activity}'[s] \\
&\quad \land\ \mathit{clock}' = \neg \mathit{clock}
\end{aligned}
$$

```
──────────────────────────── MODULE Engine ────────────────────────────
EXTENDS Components, Bags, Integers
VARIABLES brake
VARIABLE x, dx    output variables
────────────────────────────────────────────────────────────────────────
drive ≜ INSTANCE Drive WITH name ← "Drive", init_activity ← init_activity
halt ≜ INSTANCE Halt WITH name ← "Halt", init_activity ← FALSE
────────────────────────────────────────────────────────────────────────
C_slowdown ≜   suspend Drive and activate Halt
   ∧ drive!L_slowdown
   ∧ ActivateCurrent ∧ halt!Activate ∧ drive!Suspend
   ∧ UNCHANGED ⟨brake, x, dx⟩

C_speedup ≜   suspend Halt and activate Drive
   ∧ halt!L_speedup
   ∧ ActivateCurrent ∧ drive!Activate ∧ halt!Suspend
   ∧ UNCHANGED ⟨brake, x, dx⟩

L_exit ≜ halt!L_exit ∨ drive!L_exit

Activate ≜ drive!Activate ∧ halt!Suspend ∧ ActivateCurrent

Suspend ≜ drive!Suspend ∧ halt!Suspend ∧ SuspendCurrent
────────────────────────────────────────────────────────────────────────
Init ≜ drive!Init ∧ halt!Init
   ∧ InitSerial({"Drive", "Halt"})
   ∧ brake ∈ BOOLEAN ∧ x ∈ Real ∧ dx ∈ Real
Next ≜
   ∨ ((drive!Next ∨ halt!Next) ∧ JumpSerial({"Drive", "Halt"}))
   ∨ C_slowdown ∨ C_speedup
────────────────────────────────────────────────────────────────────────
Spec ≜ Init ∧ □[Next]_⟨brake, x, dx⟩
────────────────────────────────────────────────────────────────────────
```

Fig. 4 The TLA$^+$ specification of Engine

A component $C = A \otimes B$ composed in parallel from subcomponents A and B is equally represented by a TLA$^+$ module $\mathcal{M}_C = \langle Dcl_C, Def_C \rangle$ with a reachability graph $\mathcal{G}_C = \langle N_C, E_C, D_C \rangle$ induced by the specification $Init \wedge \Box[Next]_{vars}$ if the following conditions are met:

- A set of module variables includes all I/O variables that appears in both of its subcomponents; that is, for every $v^T \in V_A^{in,out} \cup V_B^{in,out}$ there exists $v \in Dcl_C$ such that $\models_{\mathcal{G}_C} v \in T$.
- The meaning of next-state action $Next$ agrees with the predicate $\varphi_A^{jump} \wedge \varphi_B^{jump}$; that is, for execution $\mathbf{w} = w_0 \dots w_n$
  - if $(a, \mathbf{w})$ is an execution of $A \otimes B$ then there is a sequence of states $\hat{p}_0 \dots \hat{p}_n$ such that $(\hat{p}_i, \hat{p}_{i+1}) \models_{\mathcal{G}_C} Next$ for $0 \leq i < n$. Location a is active at least at states $\hat{p}_0$ and $\hat{p}_n$.
  - if $(a, \mathbf{w}, b)$ is an execution of $A \otimes B$ then there is a sequence of states $\hat{p}_0 \dots \hat{p}_n$ such that $(\hat{p}_i, \hat{p}_{i+1}) \models_{\mathcal{G}_C} Next$ for $0 \leq i < n$. Location a is active at state $\hat{p}_0$ and location b is active at state $\hat{p}_n$.

  Moreover, it has to hold for both cases that values of every variable of $v \in V_{A(J)}$ in both interpretations equal; that is, $D(\hat{p}, v) = p[v]$.

We decided to reflect the component hierarchy in structure of TLA$^+$ modules although for meeting the same interpretation the other options are possible too. However, this approach enables us to structure TLA$^+$ specification in the same manner as Masaccio models. A state space of a composed component is generated according the initial predicates and next-state actions of its subcomponents. The conjunction of next-state actions requires that there are simultaneous jumps in each of the subcomponent. Moreover if one of the subcomponent reaches its end location, which causes that such component has not enabled action, it is not possible to execute any jump in any of the contained components. This configuration is then recognized as the end location of the component. The identification of end locations in behaviors is important for serial composition of components as the end locations determine the moments when currently active component looses the control and passes it to another component.

4.3 Specifying Serial Composition of Components

The serial composition of components requires that only one contained component has control at a time. This needs to be reflected in a location configuration. To enable the passing of control between components, specific actions that modify only activity array are added into the specification. Their purpose is similar to that of connector elements that can be found in architecture description languages, e.g. [17]. The example of a component composed in series is shown in figure 4. The module *Engine* instantiates two subcomponents, namely *Drive* and *Halt*. Constants *init_activity* are used to specify which component has focus initially. Component *Halt* can get focus only through interface location *slowdown* that is not accessible outside *Engine* component. Component *Drive* will get focus if component *Engine* gets one. Lines 9-12 and 14-17 contain definition of connectors that stand for transfer of control between *Drive* and *Halt* components as depicted by slowdown and speedup component interconnection in figure 1. Connector action *C_slowdown* is enabled if *Drive* component is at *slowdown* location, which is tested by *L_slowdown* predicate introduced in *Drive* module (see figure 3, line 19).

The connector activates *Halt* component and keep the component *Drive* at the end location. Also, it activates itself by including *Activate* in the conjuction of the next-state action. Predicate *Activate* simply sets the corresponding element of *activity* array to TRUE without the modifying *clock* variable:

$$
\begin{aligned}
&\mathit{Activate} \triangleq \quad \text{Activation of the current component.}\\
&\quad \land\ \mathit{activity}' \in [\mathit{Components} \rightarrow \text{BOOLEAN}\ ]\\
&\quad \land\ \mathit{activity}'[\mathit{current}] = \text{TRUE}\\
&\quad \land\ \text{UNCHANGED}\ \langle \mathit{clock} \rangle
\end{aligned}
$$

Predicate *Suspend* is included in the action predicate in order to fully define *activity* array in the next state. Without this predicate, *activity*[*Drive*] can be arbitrary boolean value. Instead, elements of *activity* array that defines states of component *Drive* and its subcomponents are set to UNDEF . An example sequence demonstrating components switching is show in fig 5. The state of the system is represented only by the actual value of *activity* array. Initially, component *Drive* is active. In the second step, the component *Drive* enters its end location, which is forced by component *DriveBrake*. This leads to inactivation of *Drive* and *Engine* components too. Component switching is done in the transition between the second and third step. Component *Halt* is activated, which allows to set *Engine* component to active state too. The state of *Drive* component set as suspended. Note that *clock* variables remains unchanged during the switch step.

In TLA$^+$ specification, the end locations can be identified by reading *activity* array. The presence of an atomic component A in its end location can be checked by expression $\mathit{activity}[A] = \text{FALSE}$. In a specification of component C obtained by parallel compositions of components $\mathbf{A} = A_1, \ldots A_n$, reaching the end location by one of the subcomponents is propagated to the top level, which is encoded as $\mathit{activity}[C] = \forall\, s \in \mathbf{A} : \mathit{activity}[s]$. Serial composition C of components $\mathbf{A} = A_1, \ldots A_n$, reaching the end location by subcomponents is propagated to the top level, which is encoded as $\mathit{activity}[C] = \exists\, s \in \mathbf{A} : \mathit{activity}[s]$. This means that all the components must be inactive in order to mark the serial composition as inactive too.

Once a component reaches its end location it should remain there until the control flow enter the component again via one of its entry interface location. In other words, the component waits in this state for the resume action. The composability requires that the component does permit the evolution of its environment (or container component) and therefore the global clocks are allowed to run, while the component is at the end location.

A component $C = A \oplus B$ composed in serial from subcomponents A and B is equally represented by a TLA$^+$ module $\mathcal{M}_C = \langle Dcl_C, Def_C \rangle$ with a reachability graph $\mathcal{G}_C = \langle N_C, E_C, D_C \rangle$ induced by the specification $\mathit{Init} \land \Box[\mathit{Next}]_{vars}$ if the following conditions are met:

- A set of module variables includes all I/O variables that appears in both of its subcomponents; that is, for every $v^T \in V_A^{in,out} \cup V_B^{in,out}$ there exists $v \in Dcl_C$ such that $\models_{\mathcal{G}_C} v \in T$.
- The meaning of next-state action *Next* agrees with the predicate $\varphi_A^{jump} \vee \varphi_B^{jump}$; that is, for execution $\mathbf{w} = w_0 \ldots w_n$
  - if $(a, \mathbf{w})$ is an execution of $A \oplus B$ then there is a sequence of states $\hat{p}_0 \ldots \hat{p}_n$ such that $(\hat{p}_i, \hat{p}_{i+1}) \models_{\mathcal{G}_C} \mathit{Next}$ for $0 \leq i < n$. Location a is active at least at states $\hat{p}_0$ and $\hat{p}_n$.
  - if $(a, \mathbf{w}, b)$ is an execution of $A \oplus B$ then there is a sequence of states $\hat{p}_0 \ldots \hat{p}_n$ such that $(\hat{p}_i, \hat{p}_{i+1}) \models_{\mathcal{G}_C} \mathit{Next}$ for $0 \leq i < n$. Location a is active at state $\hat{p}_0$ and location b is active at state $\hat{p}_n$.

  Moreover, it has to hold for both cases that values of every variable of $v \in V_{A(J)}$ in both interpretations equal; that is, $D(\hat{p}, v) = p[v]$.

Execution in a serial component consists of possibly alternating finite sequences of jumps. Each sequence belongs to behavior of some subcomponent. This principle is reflected in *Next* predicate of a composite component which is defined as a disjunction of *Next* predicates of the subcomponents. The other possibility is to have *Next* predicate defined as a conjunction of subcomponent's *Next* predicates and adding a distinguished action to all atomic components, which is enabled if the

$$\begin{bmatrix} \mathit{DriveBrake} \mapsto \text{TRUE} \\ \mathit{Accelerate} \mapsto \text{TRUE} \\ \mathit{Drive} \mapsto \text{TRUE} \\ \mathit{HaltBrake} \mapsto \text{UNDEF} \\ \mathit{Decelerate} \mapsto \text{UNDEF} \\ \mathit{Halt} \mapsto \text{UNDEF} \\ \mathit{Engine} \mapsto \text{TRUE} \\ \mathit{clock} = \text{TRUE} \end{bmatrix} \rightarrow \begin{bmatrix} \mathit{DriveBrake} \mapsto \text{FALSE} \\ \mathit{Accelerate} \mapsto \text{TRUE} \\ \mathit{Drive} \mapsto \text{FALSE} \\ \mathit{HaltBrake} \mapsto \text{UNDEF} \\ \mathit{Decelerate} \mapsto \text{UNDEF} \\ \mathit{Halt} \mapsto \text{UNDEF} \\ \mathit{Engine} \mapsto \text{FALSE} \\ \mathit{clock} = \text{FALSE} \end{bmatrix} \rightarrow \begin{bmatrix} \mathit{DriveBrake} \mapsto \text{UNDEF} \\ \mathit{Accelerate} \mapsto \text{UNDEF} \\ \mathit{Drive} \mapsto \text{UNDEF} \\ \mathit{HaltBrake} \mapsto \text{TRUE} \\ \mathit{Decelerate} \mapsto \text{TRUE} \\ \mathit{Halt} \mapsto \text{TRUE} \\ \mathit{Engine} \mapsto \text{TRUE} \\ \mathit{clock} = \text{FALSE} \end{bmatrix}$$

Fig. 5 A trace of component switch behavior

component is suspended. This activity states that the component does not engage in system behavior.

4.4 Composition Rules

In the following, we generalize the previously informally defined compositional operations to the TLA$^+$ composition rules and shows that they obey the required properties.

Definition 1 (Rule of Parallel Composition) For any set C, if

$$(\forall k \in C : v'_k = v_k) \equiv (v' = v)$$

then

$$(\otimes_{k \in C} : \mathit{Init}_k \wedge \Box[\mathit{Next}_k]_{\langle vars_k \rangle}) \equiv \\ \wedge (\bigwedge k \in C : \mathit{Init}_k) \wedge \mathit{InitParallel}(C) \\ \wedge \Box[(\bigwedge k \in C : \mathit{Next}_k) \wedge \mathit{JumpParallel}(C)]_{\langle vars_k \rangle}.$$

The rule of parallel composition gives a hint on how to create a joint specification by composing specifications of components in set C such that their instanteneous state changes are performed as joint actions. This rule is very close to the principles defined in [2]. The composition is reduced to conjunction of components specifications.

Definition 2 (Rule of Serial Composition) For any set of components C and a set of connectors S between components of C, if

$$(\forall k \in C : v'_k = v_k) \equiv (v' = v)$$

then

$$(\oplus_{k \in C} : \mathit{Init}_k \wedge \Box[\mathit{Next}_k]_{\langle vars_k \rangle}) \equiv \\ \wedge (\bigwedge k \in C : \mathit{Init}_k) \wedge \mathit{InitSerial}(C) \\ \wedge \Box \begin{bmatrix} \vee \begin{pmatrix} \wedge (\bigvee k \in C : \mathit{Next}_k) \\ \wedge \mathit{JumpSerial}(C) \end{pmatrix} \\ \vee (\bigvee j \in S : F_j) \end{bmatrix}_{\langle vars_k \rangle}.$$

The rule of serial composition catch an aspect of switching constitutent components in behaviors of the composite component. The basic idea is that composition can be reduced to disjunction of steps that can be performed by constitutent components. Nevertheless, this solely is not sufficient as we need to be sure that only one component is running at a time. To do this, an explicit control flow mechanism is introduced by means of $\mathit{activity} \in [\mathit{Components} \rightarrow \text{TRISTATE}]$ vector and connector set S. A connector is responsible for activate a selected consitutent component according to specified activation rules. A connector that can switch from component i to component j is written in the form

$$F_{ij} \triangleq L_i \wedge \mathit{Activate}(j) \wedge (\bigwedge k \in C \setminus j : \mathit{Activate}(k)),$$

where state formula L_i specifies the end location of component i. Reaching this location by a component i triggers switch action of connector F_{ij} yielding the connector j in active mode and all others in the suspended mode.

5 Conclusion

In this paper, an overview of the method capable of formal specifying and verifying embedded control systems has been presented. The method is based on the TLA$^+$, which is equipped with a highly expressive language allowing for writing clear and readable specifications. An accompanying tool, the TLC model checker, can be employed to show that the specification implies the intended properties. We illustrated the method on a simple example and provided a formal description of composition rules.

In addition to clarification of basic properties of the component model and specification method, the perspectives for future research were revealed:

- Deeper understanding of the assume-guarantee refinement in the TLA$^+$ specification framework is required and the proof that these specifications obey assume-guarantee principle as specified for Masaccio model should be given. It enables to apply the formal verification techniques to construct proofs of design correctness for individual components and their compositions.
- Specification of hybrid systems as proposed by Masaccio was not fully addressed in this paper. In particular, continous components and related issues were not considered at all. As shown in [15], TLA$^+$ is expressive enough to capture a large class of hybrid systems specifications. The question is whether

the verification can be adequately supported by the tools available for TLA$^+$.
- A refinement relation that has been defined for conjoining compositions of specifications [1] shows the capability of TLA$^+$ to develop the system implementation from an initial high-level to a high-level timed program (e.g. [8] and [6]) in a step-wise manner.

The formal model and the presented specification method is suitable, in particular, for application to the domain of distributed time-triggered systems [13] or to support design methods based on architecture description languages with formal reasoning tools.

Acknowledgements The research has been supported by the Czech Ministry of Education in the frame of Research Intentions MSM 0021630528, and by the Grant Agency of the Czech Republic through the grants GACR 201/07/P544 and GACR 102/08/1429. We thank Miroslav Sveda, Petr Matousek, Tom Hilburn, and anonymous reviewers for suggestions to improve this paper.

References

1. M. Abadi and L. Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132, 1993.
2. M. Abadi and L. Lamport. Conjoining specifications. Research Report 118, Digital Equipment Corporation, 1993.
3. Martín Abadi and Stephan Merz. An abstract account of composition. In J. Wiedermann and P. Hajek, editors, *Mathematical Foundations of Computer Science*, volume 969 of *Lecture Notes in Computer Science*, pages 499–508, Prague, Czech Republic, 1995. Springer-Verlag.
4. Patrick Cousot and Radhia Cousot. Verification of embedded software: Problems and perspectives. *Lecture Notes in Computer Science*, 2211:97–114, 2001.
5. Razvan Diaconescu, Kokichi Futatsugi, and Shusaku Iida. Component-based algebraic specification and verification in cafeobj. In *FM'99 – Formal Methods, volume 1709 of Lecture Notes in Computer Science*, pages 1644–1663. Springer, 1999.
6. Arkadeb Ghosal, Thomas A. Henzinger, Daniel Iercan, Christoph Kirsch, and Alberto L. Sangiovanni-Vincentelli. Hierarchical timing language. Technical Report Technical Report No. UCB/EECS-20, EECS Department, University of California, Berkeley, May 2006.
7. Thomas A. Henzinger. Masaccio: A formal model for embedded components. In *TCS '00: Proceedings of the International Conference IFIP on Theoretical Computer Science, Exploring New Frontiers of Theoretical Informatics*, pages 549–563, London, UK, 2000. Springer-Verlag.
8. Thomas A. Henzinger, Benjamin Horowitz, and Christoph Meyer Kirsch. Giotto: A time-triggered language for embedded programming. *Lecture Notes in Computer Science*, 2211:166+, 2001.
9. Thomas A. Henzinger, Martin Minea, and Vinayak Prabbu. *Hybrid Systems: Computation and Control*, volume 2034/2001 of *Lecture Notes in Computer Science*, chapter Assume-Guarantee Reasoning for Hierarchical Hybrid Systems, pages 275–290. Springer Berlin / Heidelberg, January 2001.
10. Peter Herrmann, Gunter Graw, and Heiko Krumm. Compositional specification and structured verification of hybrid systems in ctla. In *In Proc. 1st IEEE International Symposium on Object-oriented Real-time distributed Computing*, pages 335–340. IEEE Computer Society Press, 1998.
11. Jozef Hooman. A compositional approach to the design of hybrid systems. In *Hybrid Systems*, pages 121–148, London, UK, 1993. Springer-Verlag.
12. Michael Kaminski and Yael Yariv. A real-time semantics of temporal logic of actions. *Journal of Logic and Computation*, 13(6):921–937, 2001.
13. Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*, volume 395 of *The Springer International Series in Engineering and Computer Science*, chapter The Time-Triggered Architecture, pages 285–297. Springer Netherlands, 2002.
14. Ron Koymans, R. K. Shyamasundar, Willem P. de Roever, Rob Gerth, and S. Arun-Kumar. Compositional semantics for real-time distributed computing. *Inf. Comput.*, 79(3):210–256, 1988.
15. Leslie Lamport. Hybrid systems in tla$^+$. In *Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science*, pages 77–102. Springer, 1992.
16. Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Professional, 2003.
17. Kung-Kiu Lau, Vladyslav Ukis, Perla Velasco, and Zheng Wang. A component model for separation of control flow from computation in component-based systems. *Electronic Notes in Theoretical Computer Science*, 163(1):57–69, September 2006.
18. Edward A. Lee. Embedded software. *Advances in Computers*, 56:56–97, 2002.
19. Nancy Lynch, Roberto Segala, and Frits Vaandrager. Hybrid i/o automata. pages 496–510. Springer-Verlag, 1996.
20. Y. S. Ramakrishna and R. K. Shyamasundar. A compositional semantics of esterel in duration calculus. In *In Proc. Second AMAST workshop on Real-time Systems: Models and Proofs, Bordeux*. Springer-Verlag, 1995.
21. Ondrej Rysavy and Jaroslav Rab. A component-based approach to verification of embedded control systems using tla. In *IEEE Proceedings of International Multiconference on Computer Science and Information Technology*, pages 719–725. IEEE Computer Society Press, 2008.