In Proceedings of the *1st IEEE International Symposium on Object-oriented Real-time distributed Computing*, pages 335–340, IEEE Computer Society Press, 1998

Compositional Specification and Structured Verification of Hybrid Systems in cTLA

Peter Herrmann, Günter Graw, and Heiko Krumm
Universität Dortmund, Fachbereich Informatik, D-44221 Dortmund
{herrmann|graw|krumm}@ls4.cs.tu-dortmund.de

Abstract

Many modern chemical plants have to be modelled as complex hybrid systems consisting of various continuous and event-discrete components. Besides of the modular and easy-to-read specification, the formal verification of required properties (e.g., safety properties) is a major problem due to the complexity of the models. In practice, mostly informal argumentations exist which show that certain properties hold. The informal argumentation for one specific property does not deal with the complex system model as a whole but considers specific parts and aspects only. Our approach supports formal proofs which correspond to the informal argumentations even with respect to the use of subsystems only. It is based on the specification language cTLA supporting modular descriptions of hybrid systems. We outline cTLA and introduce the approach by means of a hybrid example system.

1. Introduction

Compositional specifications describe systems guided by their internal structure. They state the set of components, define their coupling, and refer to modular component specifications (cf., [2, 5]). Our specification technique cTLA supports not only compositionality in this usual sense. Moreover, it ensures that properties of components and subsystems are properties of the system as a whole, too [4]. Therefore so-called structured verification can be applied [3]. The proof that a certain property holds in a system is reduced to the proof of the existence of a subsystem where the property holds. If the system specification has a suitable structure, relatively small subsystems can be used and the verification therefore is facilitated substantially.

The structure of a system specification, however, is usually not well-suited to structured verification since it is implementation-oriented. Components correspond directly to physical parts of the real system and interesting system properties depend on the cooperation of many parts. Therefore cTLA supports so-called constraint-oriented specification structures (cf. [10]). Modular components of a system specification can be used to model logical constraints of system parts. Thus, the structured verification can be supplied by small subsystems consisting of those constraints only which are necessary to prove a property of interest.

Our approach is based on cTLA. cTLA stands for compositional TLA and is an extension of the Temporal Logic of Actions TLA [9]. As in TLA, systems are modelled by state transition systems and a specification has the character of a linear time temporal logic formula which describes relevant properties of the system. Moreover, cTLA recognizes the TLA-extensions to realtime systems [1] and hybrid systems [8]. In difference to TLA, cTLA supports the explicit notion of processes. Processes act as modular specification components and can represent implementation parts as well as logical constraints. Comparable to objects in DisCo [6], processes encapsulate state components and interact via joint actions.

In the sequel we give a short outline of cTLA first. Thereafter we introduce a hybrid example system and describe its compositional specification. We sketch the informal argumentation of a safety property and outline the corresponding formal structured verification. The work is funded by the German research foundation DFG.

2. cTLA

cTLA-specifications are notations for canonical TLA-formulas describing safety and liveness properties of state transition systems. However, since the syntax of cTLA is oriented at programming languages and since the canonical parts of TLA-formulas are omitted, the formula character of cTLA-specifications is not directly visible. The specifications are structured into modular definitions of process types. Instantiations of the types form processes. A process may either describe a simple process or a subsystem. A simple process directly refers to a state transition system.

```
PROCESS ControlB5Vaporize
  (vsminV13 : Real; ! min. vol. stream
   ! in the cooling of K1
   volminB5 : Real) ! min. vol. in B5

BODY
  VARIABLES
    st : {1, 3, 4, 5, 6};
    ! available states of the sub-process
  INIT ≜ st = 1;  ! initial state 1
  ACTIONS
    B5filled ≜ st = 1 ∧ st′ = 3;
      ! B5 is filled (start cooling of K1)
    CoolingWorking(pmfi : Real) ≜
      st ∈ {3,5} ∧ pmfi > vsminV13 ∧
      st′ = 4;
      ! Cool. vol. stream (pmfi) > vsminV13
    CoolingFailure ≜ ...;
    MinimumWater(pv : Real) ≜ ...;
    ConcentrationReached ≜ ...;
    TerminateProgram ≜ ...
  WF: ConcentrationReached;
END
```

As an example we show the process type *ControlB5Vaporize*. The header declares the type name and generic module parameters. The body defines the state transition system which corresponds to an instantiation of the type. The state space is defined by the state variable `st`. The initial condition `INIT` is a condition on state variables and defines the set of starting states. Finally, the body declares actions each describing a set of state transitions. An action can have action parameters. It is a condition on parameters and state variables where the state variables can also occur in so-called primed form (e.g., `st'`). The primed variables refer to the successor state of a transition. The union of the actions forms the next state relation of the process. In the course of time, a process may perform action steps (i.e., it changes its state in accordance with an action) or stuttering steps (i.e., it does not change its state while the environment performs a state transition).

Processes of a form like *ControlB5Vaporize* describe safety properties. Liveness properties are described by additional fairness assumptions on actions, e.g., `WF: ConcentrationReached` states that the action `ConcentrationReached` has to be performed weak fairly. Fairness forces the activity of a process. A fair action cannot be enabled for an infinite period of time without being executed. Weak fairness forces the execution if the action would be enabled incessantly otherwise. Strong fairness forces the execution even if the action is disabled sometimes. Unlike TLA, cTLA provides for conditional fairness assumptions to ensure the consistency of process compositions: In cTLA, a fairness statement refers to periods of time where an action is enabled as well as the environment of the process is ready to tolerate the action.

According to [1], realtime is represented by means of a real-valued state variable `now` which is incremented lively by a clock action `tick`. Unlike other variables which are private to exactly one process, `now` can be read by all processes of a system. The clock variable forms the basis for the definition of realtime and continuous properties. Additional constructs describe activity retarding and activity forcing realtime constraints. In accordance to [8], one can specify minimum waiting times and maximum reaction times for actions. Comparable to the distinction between weak and strong fairness, one may refer to volatile and to persistent enabling periods of an action. Also, the maximum reaction time construct is conditional. It forces an action only with respect to periods where the action is enabled and the environment does not block it. The notation is similar to that of fairness constraints (keywords `persistent` resp. `volatile` and `min time` resp. `max time` followed by a list of action names).

Continuous properties of a process are expressed by means of an action with the special name `CONT`. All `CONT`-actions of all processes of a system and the $tick$-action of the clock are assumed to occur simultaneously. Thus, the series of `CONT`-steps of the system approximates the continuous behaviour (cf. [8]). Usually, the `CONT`-actions contain difference equations for continuous state variables (`now'` − `now` expresses the time difference modelled by an execution of `CONT`). The inputs and outputs of continuous processes are modelled by action parameters.

```
PROCESS TankMassOfWaterAndSteam
! Mass of water / steam in tank

BODY
  VARIABLES
    vm : Real; ! mass of water / steam [kg]
  INIT ≜ vm = 0; ! initially: empty tank
  ACTIONS
    CONT (INPUT mfi, mdr : Real;
          OUTPUT m : Real) ≜
      ! mfi : mass of water / steam input
      ! mdr : mass of water / steam output
      ! m   : current mass in the tank
      m = vm ∧
      vm′ = vm + (mfi−mdr) * (now′−now);
END
```

As an example, the process *TankMassOfWaterAndSteam* declares a continuous real-valued state variable `vm`. The `CONT`-action declares the continuous inputs and outputs and list the difference equation for `vm`.

Several constructs stating safety, fairness, realtime, and continuous properties may be contained in the same process type definition. Thus, one can specify all relevant properties of a hybrid system component by one process. However, to support a fine-grained constraint-oriented system structure, we mainly use process types which concentrate on single properties of a single sort.

Systems and subsystems are described as compositions of concurrent processes. Each process encapsulates its variables and can change its state by atomic executions of its actions. The system state is the vector of the state variables of the processes. State transitions of the system correspond to simultaneous process actions and process stuttering steps. Each process performs either exactly one action or a stuttering step. Thus, the system actions can be defined by conjunctions of process actions and process stuttering steps. Consequently, concurrency is modelled by interleaving and the coupling of processes corresponds to joint actions. A system specification at first declares the processes of the system as instantiations of process types. Thereafter it describes the coupling of the processes by the definition of the actions of the systems. The process type *Vaporize* in Sec. 6 shows an example of a system description. It models a part of the example system introduced in the next section.

3. Example

We will explain our method with the following example [7]: A discontinuous process produces a sodium chloride solution of a desired concentration by means of mixing a higher concentrated sodium chloride solution with water. After its utilization the solution is separated into its original ingredients by vaporization again. The system is sketched in Fig. 1. From the tank B1 highly concentrated sodium chloride solution is introduced into tank B3. To dilute the solution, water is introduced form tank B2 to B3 afterwards. When the desired concentration is achieved, the solution is stored in tank B4 for further usage.

To separate the solution, it is introduced from B4 to tank B5, where as much water as necessary to produce the original highly concentrated solution is vaporized. The steam condenses in the cooled tank K1 and drops into tank B6. In B6 it is cooled to the temperature of the environment. Afterwards, the water is pumped back to the tank B2 using the pump P2. The sodium chloride solution of high concentration, produced by the vaporization, is introduced into the tank B7 where it is cooled, too. Finally, the cooled solution is pumped back to the tank B1 by means of the pump P1.

The process outlined above and additional processes as the initial production of the sodium chloride solution in B3 or the cleaning of the plant are controlled by batch processes. For instance, a batch process called R5 controls the separation of water and solution in tank B5. First, it triggers the introduction of solution from tank B4 to B5. When B5 is filled, the process starts the cooling of tank K1 and afterwards the heating of B5. During the vaporization it

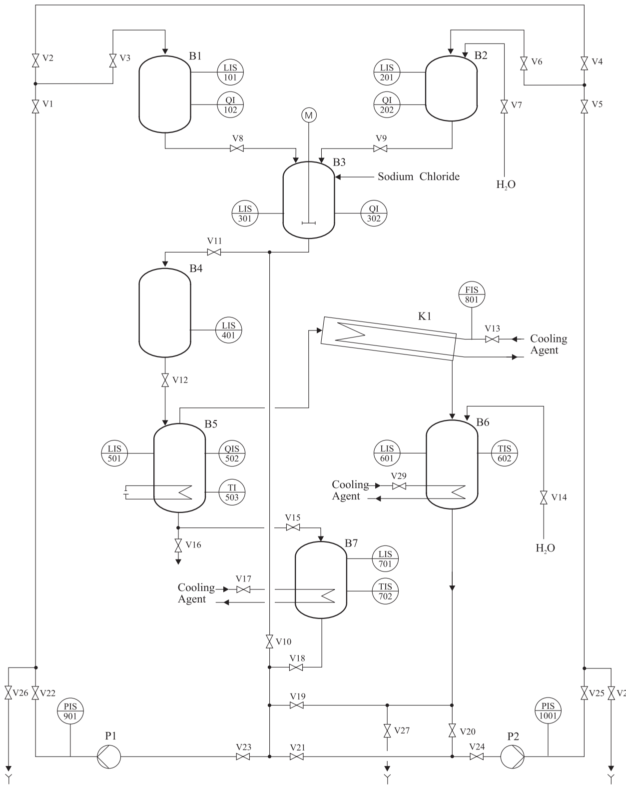

Figure 1. Sketch of the example system

constantly checks the cooling. After a failure of the cooling the heating is switched off within 0.1 seconds. The heating and cooling are switched off when the desired concentration of the solution is reached. The batch process introduces the solution into tank B7 and terminates afterwards. If the solution in B5 falls below a minimum during the vaporization, the controller switches the heating and cooling off, too. It aborts, however, without introducing the water into B7.

4. Specification

To allow the arrangement of suitable subsystem specifications, we have to develop modular specification components, each modelling single constraints of a component. For instance, the filling of a tank in our example system can be viewed by the following constraints: the mass of water and steam in the tank, the mass of dissolved sodium chloride in the tank, the specific heat energy of the water/steam in the tank, and the pressure in the tank. Thus, we specify the tank by four different cTLA-processes each describing only one constraint.

To model a pump, a valve, or an electrical heating/cooling in our example system, we use a process modelling the current actuator setting (fi. the amount of power passed to or taken from the heated/cooled tank). Furthermore, per actuator a second process describes the interface to the discrete controller. Additionally, a water heating/cooling is described by a process modelling the dependency between the volume stream of heating/cooling agent and the power supplied or removed by the heating/cooling.

The sensors of the example system detect the concentration of sodium chloride solution, the temperature of water, the amount of water in a tank, the volume stream in a pipe, and the pressure of a pump. Each sensor is modelled by four processes describing the detection, the creation of signals, the measurement, and a maximum reaction time.

The physical characteristics of the pipes linking the different tanks, actuators, and sensors are not modelled directly. Instead, processes describe the masses of water, steam, and sodium chloride running from one tank to another. The masses are calculated according to the flow formula of Torricelli, where the diameter of the pipe corresponds to the lowest diameter of all valves within the link. Furthermore, a process models the specific heat energy of the water flowing into the lower tank.

The batch process recipes controlling our example system are modelled by separate cTLA-processes, too. Moreover, since some batch processes execute a series of different tasks, one can split them into different cTLA-processes each modelling one task only. Additional cTLA-processes describe that the commands of the batch processes are executed fairly or within certain time limits.

A complete list of the specifications of the example system is available on the WWW (http://ls4-www.cs.tu-dortmund.de/RVS/P-HYSYS/).

5. Informal Argumentation

Below, we will discuss a typical safety property of the example system: Is it possible that the excess pressure, which results in the tanks B5 and K1 from the vaporization of water, exceeds a critical limit of $0.2\ Bar$? For clarification let us have a short look on the flow of steam first. The steam vaporized from water in tank B5 causes an increase of pressure. Therefore some steam runs to the tank K1 where it condenses to water again which drops into tank B6.

A violation of this safety property may occur due to two reasons. First, the difference between pressure in B5 and K1 is above the critical limit. Thus, the pressure in B5 exceeds the critical limit. Second, the systems runs too long in a critical state where the heating in B5 works but the cooling in K1 fails. Then only a part of the steam coming from B5 condenses in K1 and the pressure in K1 as well as in B5 rises.

First, we rule out that the maximum difference Δp_{max} between the pressure in B5 and K1 climbs over the limit of $0.2\ Bar$. After heating up, the heating constantly adds a power of $6000\ Watt$ to B5. Due to the enthalpy $h = 2257.9 \cdot 10^3 \frac{J}{kg}$, i.e., the energy necessary to vaporize a kilogramme of water, a mass m_{vap} of $2.66 \cdot 10^{-3} \frac{kg}{s}$ steam is created by vaporization. The pressure in B5 increases until the mass of steam running to K1 is $m_{vap} = 2.66 \cdot 10^{-3} \frac{kg}{s}$, too. Therefore we can calculate Δp_{max} on the basis of the law of Bernoulli since it is the pressure necessary to run a mass of m_{vap} steam from B5 to K1:

$$\Delta p_{max} = \frac{m_{vap}^2}{2\ \varrho\ A^2} + g\ z\ \varrho$$

Since the density of steam ϱ is equal to $0.59 \frac{kg}{m^3}$, the diameter A of the pipe linking B5 with K1 to $28.27 \cdot 10^{-6} m^2$, the difference z between the altitudes of B5 and K1 to $0.3\ m$, and the acceleration of gravity g to $9.81 \frac{m}{s}$, the value of Δp_{max} equals to the value $0.07504\ Bar$. Thus, the safety property is not violated due to the difference between the pressure in B5 and K1.

We rule out the second possibility to violate the safety property by checking that the discrete controller prevents the system to be too long in a critical state where the heating runs but the cooling fails. In a first step we will calculate the maximum time t_{max} the system may be in the critical state without the pressure in one of the tanks exceeding the limit of $0.2\ Bar$. Above, we calculated that the pressure in B5 is at most by Δp_{max} larger than that of K1. To guarantee that it does not climb over the limit, the pressure $p_{K1_{max}}$ in K1 must not exceed $0.12496\ Bar$. Next, we calculate the increase of pressure $\dot{p}_{K1}$ in tank K1 in the worst case that the cooling fails completely and a maximum mass of $m_{vap} = 2.66 \cdot 10^{-3} \frac{kg}{s}$ steam runs from B5 into K1. $\dot{p}_{K1}$ is calculated on the basis of the thermal state equation:

$$\dot{p}_{K1} = \frac{m_{vap}\ R_i\ Z\ T}{V}$$

The individual gas constant R_i of steam is equal to $461.5 \frac{J}{kgK}$, the temperature T of the steam to $373.16K$, and the volume of tank K1 to $7.854 \cdot 10^{-3} m^3$. Assuming a pressure less than $2\ Bar$ the real gas factor equals to 0.98. Thus, the value of $\dot{p}_{K1}$ is $0.5719 \frac{Bar}{s}$. The maximum time t_{max} is calculated by $\frac{p_{K1_{max}}}{\dot{p}_{K1}}$ and equals to $0.218\ s$.

In the second step we check that the disrete controller guarantees that the example system does not remain too long in the critical state, i.e., the heating is switched off, the cooling provides sufficient power, or at most t_{max} elapsed since the step into the critical state. The controller consists of batch processes (cf. [7]). However, we have to examine the batch process R5 (cf. Sec. 3) only since in the other batch processes the heating is always switched off. Of the events controlled by R5 only the vaporization may be critical since the heating is switched off during the introduction of solution from tank B4 to B5 and from B5 to B7. Furthermore, the system is not in the critical state if the cooling works properly during the vaporization. Only if the cooling fails, the system steps into the critical state. However, the batch process R5 checks the cooling constantly and reacts on a failure within $0.1\ s$ by switching off the heating. Thus, the system remains at most for $0.1\ s$ in the critical state which is shorter than the maximum time $t_{max} = 0.218\ s$ tolerated by the continuous components.

6. Formal Verification

We will point out examplarily that one can transform the informal explanations given in Sec. 5 quite easily into formal proofs. As an example we will verify that the discrete controller prevents the system to last in the critical state longer than t_{max}.

As mentioned in the introduction, we do not need the specification of the whole system to prove a property but only a subsystem specification containing only constraints relevant for the proof. We look at the informal argumentation for the property (cf. Sec. 5) and construct a subsystem *Vaporize* which contains exactly the constraint processes modelling relevant assumptions of the argumentation.

```
PROCESS Vaporize

  PROCESSES
  ! Heating B5
    B5HB : SwitchBinary;
  ! Cooling K1
    K1CP : WaterHeatingCoolerPower
           (-212 * 10^6, "direct", -10);
    ! Power depends on volume flow VF
    ! (P = -212 mill. VF); decr. -10 W/sec.
  ! Sensor for volume flow to cool tank K1
    FIS801Q : SensorVolume;
    FIS801R : SensorReading;
    FIS801S : SensorSignalMin (3 * 10^-5);
    ! Signal minimum volume flow: 30 ml/sec
    FIS801T : SensorMaxTime (0.1);
    ! Signal to be generated within 0.1 sec
  ! Batch process R5 controls vaporization
    BP5V : ControlB5Vaporize;

  ACTIONS
    CONT (INPUT V13Svs, V13d;
          OUTPUT B5Hpos, K1Cpow, K1Cvs) ≜
      B5HB.CONT(B5Hpos) ∧ ...;
    B5filled ≜ BP5V.B5filled ∧ ...;
    CoolingWorking(K1Cvs : Real) ≜
      FIS801R.Reading(K1Cvs) ∧
      B5HB.Switch("on") ∧
      BP5V.CoolingWorking(K1Cvs) ∧ ...;
    CoolingFailure ≜ ...;
    MinimumWater(B5Vv : Real) ≜ ...;
    ConcentrationReached ≜ ...;
    TerminateProgram ≜ ...;
END
```

Vaporize consists only of the cTLA-processes *B5HB* modelling the state of the heating, *K1CP* modelling the dependency between the volume flow of the cooling agent and the power of the cooling, *FIS801Q*, *FIS801R*, *FIS801S*, and *FIS801T* modelling the sensor FIS801, and *BP5V* modelling a sub-process of the batch process R5 which controls the vaporization. *BP5V* is an instance of the process *ControlB5Vaporize* listed above.

As pointed out, the system has always to fulfill the condition that either the heating is switched off, the cooling supplies sufficient power, or the system lasts in the critical state shorter than $t_{max} = 0.218\ s$. We describe this invariant condition by the formula I:

$$\begin{aligned} I \triangleq\ & \texttt{B5HB.st} = \texttt{"off"} \vee \\ & \texttt{K1CP.vpow} < -6000 \ \vee\ t_{fail} < 0.218 \end{aligned} \tag{1}$$

The variable `st` of the cTLA-process *B5HB* specifies the state of the heating. The variable `vpow` of the process *K1CP* describes the power supply of the cooling[1]. t_{fail} is an auxiliary variable[2] which indicates the time happened since the failure of the cooling. In the action `CONT` it is set to 0 when the volume flow of the cooling agent falls below $3 \cdot 10^{-5} \frac{m^3}{s}$. The proof that the formula I always holds corresponds to the TLA-theorem

$$Vaporize \Rightarrow \Box I \tag{2}$$

It means that the subsystem *Vaporize* implies that I is always valid, i.e., I is an invariant of *Vaporize*.

Yet, we cannot prove I directly but need a stronger invariant $I_h \triangleq I_1 \wedge I_2 \wedge I_3$ where

$$\begin{aligned} I_1 \triangleq\ & \texttt{BP5V.st} \neq 4 \Rightarrow \texttt{B5HB.st} = \texttt{"off"} \\ I_2 \triangleq\ & \texttt{BP5V.st} = 4 \Rightarrow \\ & (\texttt{K1CP.vpow} < -6000 \vee t_{fail} < 0.1) \\ I_3 \triangleq\ & \texttt{K1CP.vpow} \geq -6000 \Rightarrow \\ & \texttt{FIS801R.val} < 3 \cdot 10^{-5} \end{aligned} \tag{3}$$

I_h reflects that in our informal argumentation we assumed some implicit conditions which have to be proven formally now. By I_1 we describe that only the vaporization phase (controller is in state 4) can be critical since otherwise the heating is switched off. I_2 corresponds to the condition that during the vaporization either the cooling works or after its failure at most 0.1 seconds passed since otherwise the controller had switched off the heating already. I_3 reflects the assumption that the failure of the cooling is indicated by the sensor FIS801.

It is obvious that the conjunct of I_1 and I_2 implies the invariant I. Thus, it is sufficient to prove the theorem

$$Vaporize \Rightarrow \Box I_h \tag{4}$$

[1] The negative value indicates that heat energy is removed from K1.

[2] Auxiliary variables are used for verification purposes only and do not influence the behaviour of a system.

Typically invariants are verified in two steps: The invariant must hold initially and the actions of the system must not falsify the invariant.

First, we verify that I_h holds in the inital state. Since initially the heating is switched off (`B5HB.st = "off"`) and the controller is in the state 1 (`BP5V.st` $=1$), I_1 and I_2 hold. I_3 is also true since the sensor FIS801 detects no volume flow of the cooling agent in the beginning (`FIS801R.val` $= 0$).

Second, we prove that performing the joint actions of the subsystem specification *Vaporize* keeps I_h. Exemplarily we outline the proof that the action `CoolingWorking` does not falsify I_h which corresponds to the formula

$$I_h \wedge \texttt{CoolingWorking} \Rightarrow I'_h \tag{5}$$

In $I'_h \triangleq I'_1 \wedge I'_2 \wedge I'_3$ the variables occur in the primed form (e.g., `B5HB.st'`; cf. Sec. 2). It describes that the invariant holds in the state after performing the action `CoolingWorking`.

I'_1 is trivially true since by `CoolingWorking` the controller enters the state 4 and the left side of the implication in I'_1 is false. To prove that I'_2 holds, we must take into consideration that I_h (in particular I_3) is true before performing the action. `CoolingWorking` is only enabled if the volume flow of the cooling agent is sufficient (`FIS801R.val` $\geq 3 \cdot 10^{-5}$). Due to I_3 in this state the power of the cooling is also sufficient (`K1CP.vpow` < -6000). Since the action does not alter the variable `K1CP.vpow`, the right side of the implication in I'_2 is true. I_3 is considered in the verification of I'_3, too. Since I_3 holds before the execution and `CoolingWorking` does not alter the variables of the processes *K1CP* and *FIS801R*, I'_3 is also true.

In the same way we prove that the other actions of the subsystem specification *Vaporize* keep I_h. Therefore I_h and I are always true in the subsystem modelled by *Vaporize*. Since the invariant proof addressed the real time property of the action `CoolingFailure`, we cannot generalize this proof to the whole example system directly. Furthermore, we have to ensure that the action `CoolingFailure` is not blocked by the environment of *Vaporize*. Otherwise the verification that `CONT` keeps the invariant I_h might be spoiled due to the conditional character of realtime properties. However, this proof is trivial since the other processes of the system either participate to `CoolingFailure` with actions which are always enabled or with stuttering steps.

7. Concluding Remarks

We reported on the approach of the cTLA-based compositional specification and structured verification of hybrid systems and exemplified the central principles of constraint-oriented specification structures and the relatively direct formalization of informal argumentations. As the example showed, the formal modelling and verification of practical systems is feasible if the system is well-understood and consistent informal argumentations exist. To provide for further support, our current work considers the re-use of specification and verification elements.

References

[1] M. Abadi and L. Lamport. An old-fashioned recipe for real time. In J. W. de Bakker et al., editors, *Real-Time: Theory in Practice*, in LNCS 600. Springer, 1991.

[2] R. Alur, C. Courcoubetis, T. A. Henzinger, and P.-H. Ho. Hybrid Automata. In R. L. Grossman et. al., editors, *Hybrid Systems*, in LNCS 736, pages 209–229. Springer, 1993.

[3] P. Herrmann and H. Krumm. Compositional Specification and Verification of High-Speed Transfer Protocols. In S. T. Vuong and S. T. Chanson, editors, *Protocol Specification, Testing, and Verification XIV*, pages 339–346, Vancouver, 1994. IFIP, Chapman & Hall.

[4] P. Herrmann and H. Krumm. Specification of Hybrid Systems in cTLA+. In *5th International Workshop on Parallel & Distributed Real-Time Systems — (WPDRTS'97)*, pages 212–216, Geneva, 1997. IEEE.

[5] J. Hooman. A Compositional Approach to the Design of Hybrid Systems. In R. L. Grossman et al., editors, *Hybrid Systems*, in LNCS 736, pages 121–148. Springer, 1993.

[6] H.-M. Järvinen, R. Kurki-Suonio, M. Sakkinen, and K. Systä. Object-Oriented Specification of Reactive Systems. In *12th International Conference on Software Engineering*, pages 63–71, Nice, 1990. IEEE.

[7] S. Kowalewski, R. Gesthuisen, and V. Roßmann. Model-based verification of batch process control software. In *IEEE Conference on Systems, Man, and Cybernetics*, pages 331–336, San Antonio, 1994. IEEE.

[8] L. Lamport. Hybrid Systems in TLA$^+$. In R. L. Grossmann et al., editors, *Hybrid Systems*, in LNCS 736, pages 77–102, Springer, 1993.

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

[10] C. A. Vissers, G. Scollo, and M. van Sinderen. Architecture and specification style in formal descriptions of distributed systems. In S. Agarwal and K. Sabnani, editors, *Protocol Specification, Testing and Verification VIII*, pages 189–204, Elsevier, 1988. IFIP.