Journal of Universal Computer Science, vol. 7, no. 1 (2001), 37-53
submitted: 1/9/00, accepted: 13/10/00, appeared: 28/1/01 © Springer Pub. Co.

An Open Software Architecture for the Verification of Industrial Controllers

Heinz Treseler
(University of Dortmund, Germany
h.treseler@ct.uni-dortmund.de)

Olaf Stursberg
(University of Dortmund, Germany
o.stursberg@ct.uni-dortmund.de)

Paul W.H. Chung
(Loughborough University, UK
p.w.h.chung@lboro.ac.uk)

Shuanghua Yang
(Loughborough University, UK
s.h.yang@lboro.ac.uk)

Abstract: The paper presents a tool architecture which supports the formal verification of logic controllers for processing systems. The tool's main intention is to provide a front-end for modelling the controller as well as the processing systems. The models are automatically transformed into representations which can be analysed by existing model checking algorithms. While the first part of the paper gives an overview of the complete architecture, the second part introduces a newly developed modelling interface: *Process Control Event Diagrams* (PCEDs) are formally defined as a suitable means to represent the flow of information in controlled processes. The transformation of PCEDs into verifiable code is described, and the whole procedure of modelling, model transformation and verification is illustrated with a simple processing system.

Keywords: Model Checking, Formal Verification, Logic Controller, Process Control Event Diagram, Tool Development.

Categories: D.2.4, D.3.1, I.6.3, I.6.5, J.0.

1 Introduction

For processing systems that involve hazardous substances it is a crucial design task to ensure **safe** operation. The exclusion of dangerous states in processing systems can easily be seen as a safety property of the system which must be fulfilled. In computer science, such an analysis or *verification* of safety properties of **formally** specified models is an established technique. Especially the method of *model checking* has been applied successfully to some systems of technical relevance, e. g. to analyse the correct design of electronic circuits or communication protocols. Nevertheless, verification techniques are merely used to check the safe design of processing systems – at least in industrial practice. An important reason for this situation is the requirement of creating appropriate formal models.

These models must describe the behaviour of the controller (which should prevent the process from reaching dangerous states) and the reaction of the process on control inputs. Furthermore, the behaviour of the process is usually of continuous nature while the safety-related control equipment mostly shows discrete dynamics.

Another problem in applying verification to controlled processing systems is the complexity of the analysis: even for relatively small systems (from an industrial point of view) a formal modelling approach leads to remarkably large systems, such that the computational effort for analysis is prohibitively high. As a consequence, a strategy of modular and structured modelling and analysis in combination with abstraction techniques (to filter out irrelevant details) is necessary to allow verification of industrial systems. This paper describes an approach and associated tool which support the user in building suitable formal models of controlled processing systems.

In the literature, only very few applications of model checking to controlled processing systems can be found. Instead of restricting to purely discrete [MPBC91] or real-time models [HTLW97], we allow continuous behaviour which is more complex than that of simple timers. Additionally, we present the only integrated method dealing with more then one controller language.

While our tool named VERDICT (*Ver*ification of *Di*screte Controllers for *Cont*inuous Processes) contains different interfaces for modelling, it is the main emphasis of this paper to illustrate a newly created front-end. It consists of an editor for so-called *Process Control Event Diagrams* (PCEDs) which have been introduced in [CY98] to model the flow of information between controller and process. By integrating PCEDs in VERDICT, we enable the use of formal verification techniques in early design phases of processing, in which the instrumentation is identified and maybe an initial (and abstract) controller logic is specified. As an essential part of the paper, a formal definition of PCEDs is introduced.

The paper is organized as follows: in Sec. 2 the software architecture of VERDICT is described. While Sec. 3 gives a detailed definition of PCEDs and explains its transformation into verifiable models, Sec. 4 illustrates the approach by applying it to a simple batch reactor. The paper ends with a conclusion.

2 Concept

In VERDICT, the user (usually a process engineer) can use graphical and textual modelling representations with which he/she is familiar, in particular block-diagrams to set up a modular structure or common programming languages to specify the control code.

2.1 Modelling paradigm: Timed Condition/Event Systems

Timed Condition/Event Systems (TCES) were chosen as underlying modelling framework since they allow the modelling of complex technical processes in a modular fashion [EKKP95]. A complex system can be split into a set of simpler subsystems, and the connection between these modules is specified by means of two characteristic type of signals:

A *condition* signal $con(\cdot)$ is a piece-wise constant, right continuous function $con : [t_0, \infty) \to Con$ with limits from the left and Con a non-empty and finite set of conditions. For example, to model a Boolean value, the set Con could be {'false', 'true'}. An event signal $eve(\cdot)$ is a function $eve : [t_0, \infty) \to Eve_0 = Eve \cup \{0\}$ such that for each finite interval $[t_1, t_2]$ there exists a finite number of points $t \in [t_1, t_2]$ with $e(t) \in Eve$ and $e(t) = 0$ otherwise. Eve is a non-empty and finite set of events. Event signals are point-wise nonzero and carry information about currently occurring state transitions, e. g. the crossing of a reactor temperature threshold.

A TCES module consists of 11 components:

$$TCES = (U, V, X, Y, Z, C, f, g, h, \Theta(C), \gamma).$$

We denote the input and output signals and the state variables of TCES modules as follows: $U = \{u_1, ..., u_{n_u}\}$ and $V = \{0, v_1, ..., v_{n_v}\}$ are the sets of input condition or input event signals, respectively. $Y = \{y_1, ..., y_{n_y}\}$ and $Z = \{0, z_1, ..., z_{n_z}\}$ are the sets of output condition or output event signals. The state variables are denoted by $X = \{x_1, ..., x_{n_x}\}$. x_0 is the initial state.

A TCES includes real-time information which is modeled by *clocks*. Clocks are reset by an input event signal or by changing the value of the state variable X and the value rises with the gradient one. The set of all clocks is called *clock space*: $C = \{c_1, \ldots, c_{n_c}\}$. A reset of a clock "$c_i := 0$", where $i \in 1 \ldots n_c$, is denoted by a function $\gamma_{c_i} : X \times X \times V \to \{0, 1\}$. The value of the clocks is evaluated by the internal clock function τ which assigns a value to each clock:

$$\dot{\tau}_i(t) = 1 \ \text{ if } \gamma_{c_i} = 0,$$
$$\tau_i(t) = 0 \ \text{ if } \gamma_{c_i} = 1, i \in 1 \ldots n_{clocks}.$$

Also here, we distinguish between *clock events* θ_j^{eve}, $j \in \ldots n_{eve}$ and *clock conditions* θ_j^{con}, $j \in 1 \ldots n_{con}$. The set of all n_{eve} clock events over the clock space C is denoted by $\Theta^{eve}(C)$ and the set of all n_{con} clock conditions is denoted by $\Theta^{con}(C)$: $\Theta(C) = \Theta^{con}(C) \cup \Theta^{eve}(C)$. A clock event is a function with a Boolean result: $\theta_j^{eve}(t) : [t_0, \infty) \to \{0, 1\}$. The value is 1 or *true* if a clock value exceeds a certain threshold. Clock events can be used to enforce transitions. They have the form "$c_i = T$" where $i \in 1 \ldots n_c$ and $T \in \mathbb{N}^+$. A clock condition is also a function with a Boolean result: $\theta_j^{con}(t) : [t_0, \infty) \to \{0, 1\}$. A Clock condition is fulfilled, i. e. its value is 1 or *true*, if the clock value is in a certain interval. It is of the form "$T < c_i \leq T'$" or "$c_i > T$" where $i \in 1 \ldots n_c$ and $T, T' \in \mathbb{N}^+$. This concept is closely related to that of Timed Automata (TAs) [AD94].

The behaviour of a TCES module is described as follows: a transition is enabled by an input condition or forced by an input event and can depend additionally on the clock values of the system. This is described by the state transition function f. State transitions are defined by $f : X \times U \times V \times \Theta(C) \to 2^X$, i. e. a new discrete state is evaluated depending on the current state, the current input signals, and the values of the clocks. The result can be a set of possible new states expressing non-deterministic behaviour. The current state, the condition input signal, and the clock condition determine the condition output according to a condition output function $g : X \times U \times \Theta^{con}(C) \to Y$. Finally, an event output function $h : X \times X \times V \times \Theta^{eve}(C) \to Z$ satisfying $x \in X : h(x, x, 0, 0) = 0$ specifies the event output depending on the current state transition, the event input signal, and the clock events. The valid sets of state, signal, and clock trajectories are given by:

$$c(t) = \tau(c(t^-), \gamma(t)),$$
$$x(t) \in f(x(t^-), u(t^-), v(t), \theta^{con}(t^-), \theta^{eve}(t)),$$
$$y(t) = g(x(t), u(t), \theta^{con}(t)),$$
$$z(t) = h(t^-, x(t), v(t), \theta^{eve}(t)).$$

The argument t^- of a function set fct denotes the left-hand limit $lim_{\delta \to 0} fct(t - \delta)$.

The output conditions or output events of a module can be connected with the input conditions or input events of another module. In contrast to the non-causal synchronization mechanisms of automata models (e. g. TAs), the condition and event signals reflect cause and effect of the interactions between different parts of the process and the controller.

2.2 Architecture

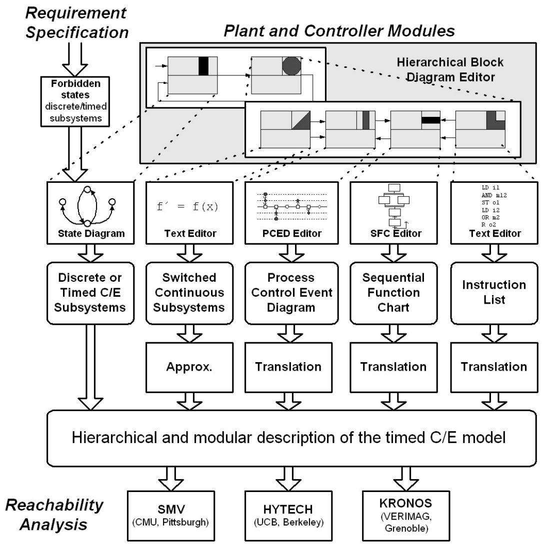

Figure 1: Software architecture of VERDICT

Fig. 1 illustrates the software architecture. It was designed to offer different alternatives on the modelling level as as well on the analysis level. The main part of the user interface consists of a hierarchical block diagram editor in which the blocks represent physical elements of the processing systems, and the exchange of information between blocks is specified by arrows. The blocks can represent purely discrete, timed-discrete dynamics or hybrid or continuous behaviour.

Blocks with discrete or timed-discrete dynamics can be specified as a state diagram or textually. The continuous dynamics can be modeled by arbitrary first-order differential equations with switched discrete inputs. Blocks with hybrid behaviour are also specified textually using a corresponding editor. The other editors which do not rely on the TCES paradigm are explained in Sec. 2.3.

All blocks which are not directly specified as TCESs are translated into this type of model. The translation of blocks with continuous dynamics additionally involves an approximation step in order to obtain verifiable TCES representations [Stu00]. For analysis, the user can choose a particular analysis tool (see Sec. 2.4).

2.3 Modelling front ends

Since it is the objective to check controllers which are designed to run on industrial control computers, i. e. Programmable Logic Controllers (PLCs), VERDICT offers the additional facility to include PLC programmes and to transfer these into TCES automatically. At present, the textual PLC language *Instruction List* and the graphical language *Sequential Function Charts* according to the standard IEC 61131-3 [IEC93] are supported by VERDICT [TBK00].

In addition to these facilities to import controller code, the so-called *Process Control Event Diagrams* (PCEDs) can be imported to VERDICT. PCEDs, which are introduced in detail in Sec. 3, were introduced by Chung and Yang to support the analysis of processing systems in early design stages with respect to the instrumentation [CY98].

2.4 Analysis

To enable the analysis of the complete TCES model by existing analysis tools, the model has to be converted into the input format of the chosen analysis tool. The tools Kronos [DOTY96] and HyTech [ACHH93] are able to analyse TAs. To integrate these tools, we take advantage of the fact that TCESs can be converted into TAs [HLU$^+$97]. Additionally, the discrete model checker SMV was chosen as the third model checker for integration. To use SMV, the infinite space of all clock values of the TCES has to be approximated by a finite automaton, which combines infinitely many time points to a discrete *region* [AD94].

A reasonable choice of an analysis tool for a specific example is obvious. For systems with a large continuous part (i. e. several clocks and timing conditions) the real-time model checkers HyTech and Kronos are appropriate. For system with very few clocks and timing constraints, the transformation into an SMV model has been found to be efficient. Certainly, SMV is the best choice to analyse purely discrete models.

3 Process Control Event Diagrams (PCEDs)

A PCED is an abstract and qualitative model of the communication between process, controller, and operator. The advantage of this representation is that connections between process variables and the control logic can be visualized in a very simple and descriptive manner. PCEDs were originally intended to support Hazard and Operability (HAZOP) studies, in which a group of engineers discusses the instrumentation layout of a processing system in order to find sources of potential hazards. Since PCEDs show which information is exchanged between process and controller, it can help to detect whether information about safety-relevant events is missing in a component, if the design of a logic controller is wrong, or if an operator action can cause dangerous situations. Due to its simplicity, PCEDs can be understood by people from different engineering domains and they can provide the basis on which the HAZOP discussions can take place. PCEDs were already successfully applied in an industrial project [RYC98]. The idea behind integrating a PCED interface into VERDICT is to use formal methods instead of informal discussions to detect the effects of instrumentation failures.

An example of a PCED is shown in Fig. 2: A PCED illustrates the interaction between nodes which are arranged on five different layers. The nodes represent the components involved in the system (e. g., sensors, actuators, control algorithms), and an edge between two nodes stands for the propagation of a signal. While PCEDs were used in an informal manner so far, we now introduce a formal definition and provide a transformation algorithm into TCESs.

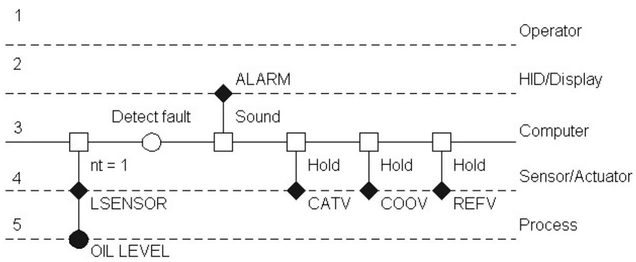

Figure 2: PCED controller specification

3.1 Syntax

A PCED consists of 11 components:

$$PCED = (Lay, Nod, \lambda, Egd, Act, PrV, \pi, OpV, \omega, RsV, \rho),$$

in which Lay is an ordered finite set containing five elements that represent the *layers*:

$$Lay = \{l_{Pr}, l_{S/A}, l_{Co}, l_{HID}, l_{Op}\}.$$

"Pr" stands for "Process", "S/A" for "Sensor/Actuator", "Co" for "Computer", "HID" for "Human Interface Device", and at last "Op" for "Operator". The layer l_{Co} is represented graphically by a horizontal solid line, and all other layers by a dotted line according to the order in Lay. Each layer is marked with its index, for example, the layer l_{Co} has the label "Computer".

Nod is a tuple of 7 different sets whose elements are called *nodes*:

$$Nod = \{Nod_{Pr}, Nod_{S/A}, Nod_{Tr}, Nod_{Co}, Nod_{Ju}, Nod_{HID}, Nod_{Op}\},$$

and each of these sets in Nod contains a finite number of nodes. The abbreviation "Tr" stands for "Transfer", respectively "Ju" for "Jump". The elements of Nod_{Pr} and of Nod_{Op} are represented graphically by a filled circle ($\bullet$), the elements of $Nod_{S/A}$ and of Nod_{HID} by a filled diamond ($\blacklozenge$), the elements of Nod_{Tr} by an empty square ($\Box$), the elements of Nod_{Co} by an empty circle ($\circ$), and the elements of Nod_{Ju} by an empty diamond ($\Diamond$). To make the identification of a specific node in the graphical representation easier, an arbitrary label can be attached to the node.

The $layoutfunction$ λ determines to which layer the different nodes are assigned, i. e. $\lambda : Lay \times Nod \to \{0, 1\}$, s.t. for $l \in Lay$, $n \in Nod_{\sim}$, where $Nod_{\sim}$ is one of the elements in Nod: $\lambda(l, n) = 1$ if

$$\begin{aligned} &([l = l_{Pr} \wedge n \in Nod_{Pr}] \ \vee \ [l = l_{S/A} \wedge n \in Nod_{S/A}] \\ &\vee [l = l_{Co} \wedge (n \in Nod_{Tr} \vee n \in Nod_{Co} \vee n \in Nod_{Ju})] \\ &\vee [l = l_{HID} \wedge n \in Nod_{HID}] \ \vee \ [l = l_{Op} \wedge n \in Nod_{Op}]) \end{aligned}$$

and $\lambda(l, n) = 0$ else.

Hence, the elements of Nod_{Co}, Nod_{Tr}, and Nod_{Ju} are placed on the layer l_{Co}, elements from the other sets in Nod are placed on the layer with the same index.

Edg is an finite set of *edges*: $Edg = \{e_1, ..., e_{n_{edg}}\}$. Each edge $e \in Edg$ defines an ordered pair of two nodes, where the following combinations are permitted:

$$\begin{aligned} &e^{I} = (n_{Pr}, n_{S/A}) \ \vee \ e^{II} = (n_{S/A}, n_{Tr}) \ \vee \ e^{III} = (n_{Tr}, n_{S/A}) \\ &\vee\, e^{IV} = (n_{Tr}, n_{HID}) \ \vee \ e^{V} = (n_{Op}, n_{HID}) \ \vee \ e^{VI} = (n_{HID}, n_{Tr}), \end{aligned}$$

with $n_{Pr} \in Nod_{Pr}, n_{S/A} \in Nod_{S/A}, n_{Tr} \in Nod_{Tr}, n_{HID} \in Nod_{HID}, n_{Op} \in Nod_{Op}$. The graphical representation of an edge is an arrow which points from the first to the second node of the pair e_i, $i \in 1 \ldots n_{edge}$. An edge of the type e^{II} is additionally labeled by an element of the set {"*none*", "*cont*", "$pt = c$" , "$nt = c$"} with $c \in \mathbb{R}$. The labels "$pt = c$" and "$nt = c$" mean that a process variable (see below) crosses a threshold c in positive or negative direction. "pt" stands for "positive threshold" and "nt" for "negative threshold", respectively. "*cont*" refers to the forwarding of a continuous variables. Furthermore, an edge of the type e^{III} is marked by a label taken from the set {"*none*", "*cont*", "*hold*", "*open*", "*close*", "*turn on*", "*turn off*", "*increase*", "*decrease*"}. All other edges can optionally be marked with the label "*none*" or with an arbitrary label.

Act denotes an ordered finite set of *actions*, which either denote a calculation within a node on the computer layer or the transmission of signals between different layers. The calculations performed within the nodes are explained in Sec. 3.2. An action a_j is defined as one of the following ordered combinations of nodes and edges:

$$\begin{aligned} a_j &= (n_{Pr}, e^I, n_{S/A}, e^{II}, n_{Tr}) \;\vee\; a_j = (n_{Tr}, e^{III}, n_{S/A}) \\ \vee\, a_j &= (n_{Tr}, e^{IV}, n_{HID}) \;\vee\; a_j = (n_{Op}, e^V, n_{HID}, e^{VI}, n_{Tr}) \\ \vee\, a_j &= (n_{Ju}) \;\vee\; a_j = (n_{Co}),\; j \in 1 \ldots n_{act}. \end{aligned}$$

The edges symbolize that a signal is sent from the node on the left to the node on the right. For the last two types, the actions represent just the calculation within the n_{Ju} or n_{Co}. The order of actions $a_j,\ j \in 1 \ldots n_{act}$ within the set Act corresponds to the order (from left to the right) with which they are arranged in the graphical representation.

$PrV = \{p_1, \ldots, p_{prv}\}$ is the set of *process variables*, where each $p_j \in PrV$ refers to one of the real-valued quantities of the process that are measured (as e. g. temperature or pressure). OpV denotes a set of n_{opv} *operator variables*, which are defined on a Boolean domain or on $\mathbb{R}$. RsV is the set of n_{rsv} output variables or *result variables*, which are as well defined on $\{0,1\}$ or $\mathbb{R}$. The functions ω, ϕ and ρ assign the variables to specific layers: $\pi : PrV \times N_{Pr} \rightarrow \{0,1\}$, $\omega : OpV \times N_{Op} \rightarrow \{0,1\}$, $\phi : RsV \times N_{HID} \cup N_{S/A} \rightarrow \{0,1\}$ (i. e. the value represents that a combination of variables and node exist). While at most one variable can be assigned to a certain node, a variable can occur at several nodes of the specific type.

3.2 Semantics

To give semantics to a PCED, we define a *run* of a PCED as a sequence of *discrete states*. Such a discrete state is introduced as a tuple:

$$sta = (a_{act}, Env, Mem)$$

in which the *environment* Env and the *memory* Mem are sets of variables of the type real or Boolean. Env contains the current values of all variables in PrV, OpV, and RsV. Mem contains copies of these variables sets, which are denoted by PrV^*, OpV^*, and RsV^*. Additionally, Mem can contain a set MrV of globally defined *marker variables*. These can be used as auxiliary variables in the computations that are carried out in the nodes assigned to lay_{Co} (see below). The variables of these four sets are denoted by : p_i^*, $i \in 1..n_{prv}$, o_i^*, $i \in 1 \ldots n_{opv}$, r_i^*, $i \in 1 \ldots n_{rsv}$, and m_i, $i \in 1 \ldots n_{mrv}$.

The separation between Env and Mem refers to the scanning mode of a PLC (see Sec. 2.3), i. e. the process variables are scanned from the sensors and written to a particular part of the memory modeled by Mem. During execution, the control software works with these variables and does not access the current values of the sensors and actuators. This construction avoid inconsistencies due to the fact that the variables in Env change their values during a cycle.

$a_{act} \in Act$ is the currently active action which can alter the values of those variables in Env and Mem that are assigned to the nodes of a_{act}. Hence, a state of the PCED is given as the combination of an action and the setting of all variables after the action is carried out. A *run* of the PCED is a (possibly infinite) sequence of states:

$$Run = (sta_1 \xrightarrow{t_1} sta_2 \xrightarrow{t_2} \ldots).$$

The computations within a state (according to an action a_{act}) and the *transitions* between two states (at the time t_k) have to be specified in more detail. If a_{act} contains an edge $e \in Edg$, data are transmitted from the first to the second node in e. If an edge is marked with the label "none", the data exchange is interrupted – this models the possibility of a transmission error. To specify the computation within a node more accurately, we assign a function f to each node $n \in Nod_{\sim}$, $Nod_{\sim} \in Nod$. Depending on the type of nodes, the following applies for the function f:

- For $n \in Nod_{Op}$, the function $f(\varnothing) = out$ has no input ($\varnothing$ is an empty symbol) and it delivers the value *out* of the variable $o_i \in OpV$ that is assigned to n. Thus, f models the behaviour of the operator. Through an edge of the type e^V, the value *out* is sent to the corresponding node $n' \in Nod_{HID}$.
- For a node $n = Nod_{HID}$ that models the human interface device, the function f depends on the type of the edge $e^{VI} = (n_{HID}, n_{Tr})$ or $e^{IV} = (n_{Tr}, n_{HID})$. In the case of e^{VI}, the output value is forwarded to the computer layer by the function f. The node models an input device:
  - if $lab =$ "*none*" then $f(in) = \emptyset$ with $\emptyset$ as a failure symbol,
  - else $f(in) = in$ with $in \in \mathbb{R}$ or $in \in \{false, true\}$.

  In the case of e^{IV}, the value in received from the computer layer is assigned to the corresponding result variable $r_i := f(in)$, and it applies:
  - if $lab =$ "*none*" then $f(in) = \emptyset$,
  - else $f(in) = in$ with $in \in \mathbb{R}$ or $in \in \{false, true\}$.
- For $n \in Nod_{Pr}$, we have $f(\varnothing) = out$ with $out \in \mathbb{R}$, and *out* represents the value of the corresponding process variable $p_i \in PrV$. The value *out* is sent to that node $n' \in Nod_{S/A}$ which is part of the edge $e = (n, n')$ of the type e^I.
- For a node $n \in Nod_{S/A}$, one has to distinguish between the cases that n represents a sensor, respectively an actuator. A *sensor node* receives its input value from a node $n_{Processor}$ and sends its output value to a node n_{Transf} on the computer layer. The label $lab \in \{$"*none*, "*cont*", "*pt* $= c$", "*nt* $= c$"$\}$ of the edge $e = (n_{S/A}, n_{Tr})$ determines the functionality of f:
  - If $lab =$ "*pt* $= c$" then $f(in) = \begin{cases} true, \text{ if } in \geq c \\ false, \text{ otherwise} \end{cases},$
  - if $lab =$ "*nt* $= c$" then $f(in) = \begin{cases} true, \text{ if } in \leq c \\ false, \text{ otherwise} \end{cases},$
  - if $lab =$ "*cont*" then $f(in) = in$.

  An *actuator node* gets its input variables from a node n_{Tr}. Also here, the label $lab \in \{$ "*none*, "*open*", "*close*", "*turn on*", "*turn off*", "*hold*", "*increase*", "*decrease*", "*cont*"$\}$ of $e = (n_{Tr}, n_{S/A})$ determines the function f. The output value is assigned to the corresponding result variable $r_i := f(in)$:

- If $lab =$ "*open*" $\vee$ $lab =$ "*turn on*" $\vee$ $lab =$ "*hold*" $\vee$ $lab =$ "*decrease*" $\vee$ $lab =$ "*increase*"
  then $f(in) := \begin{cases} true, \text{ if } in = true \\ false, \text{ otherwise} \end{cases},$
- if $lab =$ "*close*" $\vee$ $lab =$ "*turn off*"
  then $f(in) := \begin{cases} false, \text{ if } in = true \\ true, \text{ otherwise} \end{cases},$
- if $lab =$ "*cont*" then $f(in) := in$,

Thus, the output value is $true$ if a device (e. g. a valve) is opened or on, and $false$ if it is closed or off. The interventions *hold*, *decrease*, and *increase* are represented with the symbol $true$.

– For $n \in Nod_{Tr}$, one also has to distinguish between the cases that signals are received from $lay_{S/A}$ or lay_{HID} and that signals are emitted to one of these layers. Received values are used to update the copy of the operator variable or the process variable in Mem (maybe manipulated by the computation in $n \in Nod_{HID}$ or $n \in Nod_{S/A}$). Values which are send out to $n_{S/A}$ or n_{HID} are taken from Mem.

– An arbitrary control algorithm can be assigned to the nodes $n \in Nod_{Computer}$, and f can change the values in Mem. More precisely, the variables PrV^* and OpV^* can only be read and the other variables can be read and manipulated, s. t. we can write: $f : PrV^* \times OpV^* \times RsV^* \times MrV \to RsV^* \times MrV$. (Since the sets of variables may include continuous and discrete variables, we can represent continuous and discrete controller algorithms.)

– No data are transmitted or manipulated in the nodes $n \in Nod_{Ju}$ (i. e. we have $f(\emptyset) = \emptyset$).

The run of the PCED is determined by the following steps:

1. Initialize all variables and activate the action specified with sta_1. This action is called the *active action* a_{act}. Set the state index i to the value 1.
2. Evaluate the functions f of the nodes involved in a_{act} and update the values of the variables in Mem and Env (i. e. the process variables in Env are changed by the process behaviour).
3. At a point of time t_i, the state sta_i is left through a transition into state sta_{i+1}. While the values in Mem are not changed by the transition, the active action of sta_{i+1} follows from the following rule:
   – If the active action of sta_i is of the form $a_{act} = (n_{Ju})$, the next action is determined by a *jump*. The execution of a jump can depend on additional conditions, i. e. it can depend on the variables of Mem: For each node $n_j \in Nod_{Ju}$, a so-called *jump set* $Ju_j = \{ju_1, \ldots ju_{n_{ju}}\}$ exists where a jump ju_k, $k \in 1 \ldots n_{ju}$ is given as a pair $ju_k = (cn_k, a_l)$, $a_l \in Act$. If for the *condition* function cn_k applies:

$cn_k(p_1^*, \dots p_{n_{prv}}^*, o_1^*, \dots o_{n_{opv}}^*, r_1^*, \dots r_{n_{rsv}}^*, m_1, \dots m_{n_{mrv}}) = 1$ for the current values of the variables, the jump ju_k determines the action a_l as the active action of state sta_{i+1}. If $ju_k = (true, a_l)$ applies, the jump is unconditional and it takes place independently of the values of the variables in Mem. Only one of the jump conditions in Ju_j can be true at the same time, such that the determination of the next action through a jump is deterministic.

– If the action in sta_i is of one of the other types (i. e. no jump node n_{Ju} is involved), then the active action in sta_{i+1} is the next action in the ordered set Act, i. e. if a_j was active in sta_i, then a_{j+1} becomes active in sta_{i+1}.

4. Return to step 2.

The behaviour of PCED is now completely specified by Run.

3.3 Translation into TCESs

For each continuous variable in Mem or Env, we define a mapping $\delta : \mathbb{R} \rightarrow \{1, \dots, 5\}$ which gives a discrete representation x_i of the continuous variable. The value 3 represents the normal point. The values 2 and 4 denote a negative or a positive yet non-critical deviation from this point while the values 1 and 5 are critical positive or negative deviations.

The structure of the block-diagram which results from the translation is illustrated in Fig. 3. One block each is introduced to represent the process variables (PrV module), the operator variables (OpV module), and the memory Mem (memory module). Additionally, the structure contains one block for each action of the PCED (which does not involve a jump node). Referring to the type of the nodes in a_j ($n_{S/A}$, n_{HID}, n_{Co}), the modules represent the behaviour of either a sensor, an actuator, an HID, or a controller function. A further block, named *status*, is introduced to activate the other blocks in the order determined by the ordering of the set Act. If a module is activated by the appropriate event signal, it executes the function f of the corresponding node. The event signals between the blocks are marked by "lightning" symbols, and the condition signals by plain arrows. The following sections describe the different blocks in more detail.

3.3.1 PrV, OpV, and memory modules

The PrV module consists of all process variables involved in the system. Every process variable is represented as a state variable $d_i \in \{1, \dots, 5\}$. Similar to the PrV module, the OpV module represents the operator variables. These variables can also be of Boolean type. The variables of Mem are modeled by the memory module. All process variables can be read by the sensor and all operator variables by the input HID. The internal variables in the memory module can be read (as condition signals) by the display HID and the actuator. To force the memory modules to change their values, input event signals are necessary. In Fig. 4 the OpV module is shown as an example.

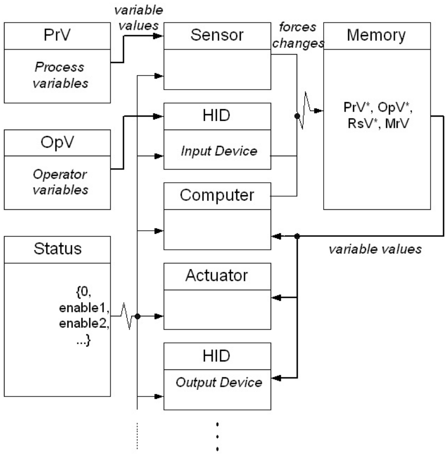

Figure 3: TCES model structure

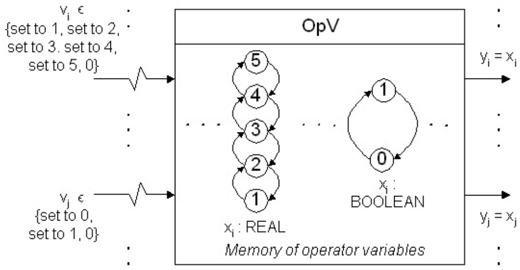

Figure 4: The OpV module

3.3.2 Sensor modules

By means of the input condition signal $u \in \{1, \ldots, 5\}$ the sensor module receives the values of the appropriate process variables and sends it to the memory module by an event output signal z. A state variable is necessary to be able to model the dynamics of the sensor and to model possible faults. The input event $v = enable$ forces the sensor to read the value of the condition input variable u and to generate the event output z.

The dynamics of the sensor module depends on the assigned label. In case of a label "*cont*", the state variable $x \in \{1, \ldots, 5\}$ gets the value of the condition input variable u. In the case of an label "$pt = c$" or "$nt = c$", $c \in \mathbb{N}$, the sensor works as a limit switch, and the state variable is defined on the set $\{0, 1\}$. The value is 1 if the threshold is crossed in the positive ($u >= c$) or negative ($u <= c$) direction, and zero otherwise. A failure can be modeled by the label "*none*", so that the activation of the sensor module does not have any effect. Fig. 5 illustrates a sensor module derived from a node $n \in Nod_{S/A}$ with the label "$nt = 1$".

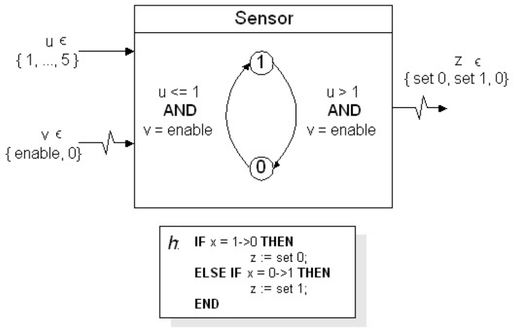

Figure 5: Sensor module

3.3.3 Actuator and HID modules

The actuator module receives its input from the memory module as condition input signal. If the assigned label lab is "*cont*" the state variable $x \in \{1, \ldots, 5\}$ gets the value from the input condition signal $u \in \{1, \ldots, 5\}$ when its activation is forced by the input event $v = enable$. If the label lab is in {"*open*", "*close*", "*turn on*", "*turn off*", "*hold*", "*increase*", "*decrease*"} the state variable x and the input condition signal u consist of the set $\{0, 1\}$.

Analogously, an HID module (serving as a display device) receives the input variable from the memory module. The state variable of actuator modules and of these HID module represent the result variables RsV. On the other hand, HID modules representing input devices receive the input from the OpV module. The output is sent to the memory module. To model failures, these modules can be provided with the label "*none*".

3.3.4 Computation modules

According to the semantics of PCEDs, computation nodes can represent arbitrary control algorithms. Since the state variable x is defined as a discrete set, in practice, we are limited to some template algorithms which may have to be refined manually. At the moment, a proportional controller and elementary algebraic function, e. g. addition and subtraction, are supported.

The process variables, operator variables, and variables of the memory module are introduced as input condition signals. The control algorithm can set the variables in the memory module to new values through the event output signals of the computation blocks.

3.3.5 Status module

The events produced from the status module correspond to transitions within the module, which are forced by a clock $tick$. This $ticks$ are generated by a clock variable that is repeatedly reset when its arbitrarily chosen threshold time is elapsed. Therefore, the module needs n_{act} transitions $t_1, \ldots t_{n_{act}}$ and $n_{act} + 1$

states. The transitions trigger the execution of the sensor, actuator, HID, or computation modules. For example, the sensor module is forced to read the values from the process variables. Jumps are considered by additional transitions, which modify the order of the activation of the actions. If the jump condition function cn is not always $true$, i. e. the jump is conditioned, the current values of the PrV, the OpV, and the memory module are fed into the status module by condition signals $u_1, \ldots, u_{n_u}$. The corresponding transition can only take place if $cn(u_1, \ldots, u_{nu}) = 1$ applies. A status module is illustrated in Fig. 6.

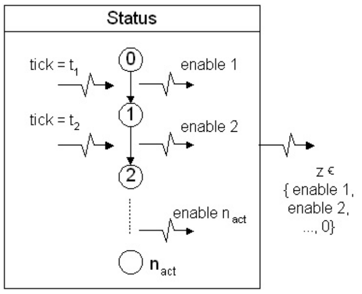

Figure 6: Status module

4 Example

4.1 A controlled batch reactor

The example system is taken from [Kle95] and is sketched in Figure 7. It mainly consists of a reactor and a condenser, in which the vapour produced in the reactor is fed. The temperature of the system is controlled by changing the flow of catalyst and cooling water. An increase of the catalyst flow intensifies the reaction, which leads to an increased temperature. The system can be cooled down by increasing the cooling flow.

The example PCED in Fig. 2 illustrates an emergency control for this batch reactor. If a fault occurs, i. e. if the measured oil level falls below a certain threshold, an alarm is displayed and the output values *catalyst flow*, *cooling flow*, and *reflux flow* are kept constant by holding the *catalyst valve* (CATV), the *cooling valve* (COOV), and the *reflux valve* (REFV). The values are hold until an operator has fixed the failure and restarts the control programme. In every case, i. e. also in the mentioned emergency situation, the controller must prevent the batch reactor from overheating. The control algorithm “detect fault” assigned to the second action of the PCED sets the copies of $Alarm^*$ (*display Alarm*), $CATV^*$ (*hold CATV*) , $COOV^*$ (*hold COOV*), $REFV^*$ (*hold REFV*) in Mem to true, if the sensor has detected a low oil level. Only after the execution of the actions including appropriate actuators, the modifications become apparent in the environment Env. In pseudo-code, the very simple control algorithm looks like:

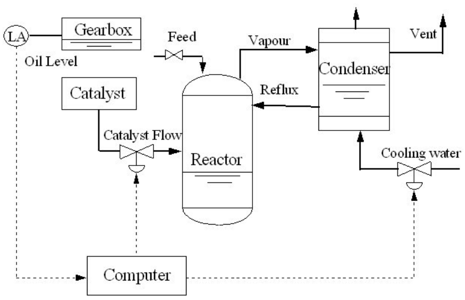

Figure 7: The example reactor

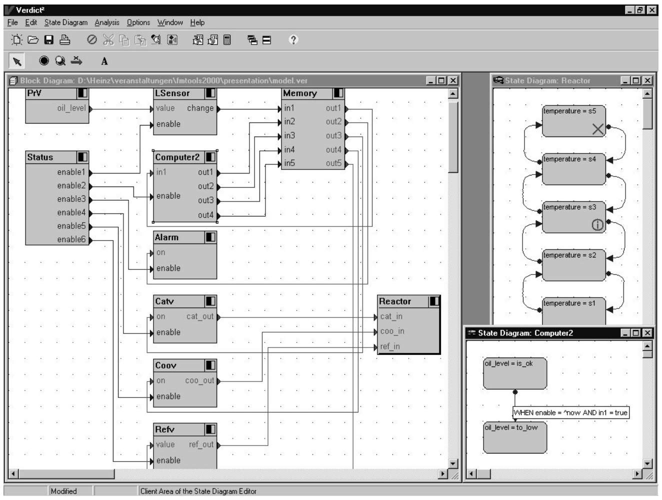

Figure 8: Screen-shot of the VERDICT model

IF $oil\_level\_to\_low$ THEN
    $Alarm^*$:= TRUE, $CATV^*$:= TRUE, $COOV^*$:= TRUE, $REFV^*$:= TRUE;
END

4.2 Modelling and model checking

To perform a safety analysis for the example, the PCED is loaded in VERDICT and translated into the equivalent TCES model automatically. The different modules and their interconnection are illustrated in Fig. 8 by a screen-shot of the VERDICT model. If the sensor module LSENSOR detects a low oil level, i. e. the input $in1$ of the COMPUTER2 module is $true$, it switches to the state *oil level to low* when activated. With this transition events are produced, which set the variables $Alarm^*$, $CATV^*$, $COOV^*$, and $REFV^*$ to $true$. The generated TCES model of the control device has to be completed manually by a simple model of the basic reactor behaviour: A module named *reactor* is added, and the values of the actuator modules CATV and COOV influence the temperature of the reactor. Therefore, the values of the actuators CATV and COOV are fed to the reactor module by condition signals. Its single state variable *temperature* represents the process variable *reactor temperature*, defined again on the set $temperature \in \{1 \ldots 5\}$. Low cooling water flow and high catalyst flow cause an increase of the reactor temperature, while high cooling water flow and low catalyst flow result in the opposite. After a response time (modeled by a clock variable) the state variable changes its value according to the input condition signals.

The safety requirement is that the batch reactor should never be overheated, i. e. that the variable *temperature* in the reactor module should never reache the value 5. In VERDICT, this forbidden state is marked by a cross in the state graph editor (see also Fig. 8), and initial states are marked with an "i".

SMV was chosen to analyse the system. The result was that the safety requirement was not fulfilled. The counter example generated by SMV makes it easy to locate why violation occured. An emergency (low oil level) can happen even when the catalyst flow is high and the cooling flow is low and the controller keeps the flows constant. In this case, the reactor temperature keeps increasing and violates the safety requirement eventually.

For a safe operation, the control logic has to be updated as follows: instead of keeping the flows constant in the case of an emergency, the catalyst valve has to be closed and the cooling valve has to be opened. This stops the reaction and prevents the reactor from overheating.

5 Conclusion

A tool environment for the formal verification of logic controllers for chemical plants was presented. The tool offers a set of graphical and textual user interfaces to create timed discrete models of the process in a block-oriented fashion, and it allows the user to include controller programmes compliant to the IEC 61131-3 standard. Continuous subsystems and controller programmes are automatically replaced by TCES models

It was shown that the architecture of VERDICT is open in the sense that new types of models used in process engineering, such as PCEDs, can be integrated. On the analysis level, we have chosen to provide interfaces to existing model checkers. The set of tools, for which interfaces are provided, enables us to choose the one which seems to be most appropriate for a specific technical system. A further successful application of VERDICT is described in [KEPS99].

The described integration of PCEDs into the VERDICT framework makes it possible to use formal methods to check the instrumentation and the controller functionality in early design stages of processing plants. Such investigations can help to detect failures in the design of discrete controllers as early as possible. This is especially important because the detection and correction of failures in the detailed control code at later design phases is a very cumbersome task.

Acknowledgments

Thanks go to the former leader of the VERDICT project, S. Kowalewski, and to S. Engell for helpful comments. The work was financially supported by the German Research Council (Deutsche Forschungsgemeinschaft, DFG) within the KONDISK-programme. The cooperation between Loughborough University and University of Dortmund was made possible by funding from the British-German Academic Research Collaboration Programme.

References

[ACHH93] R. Alur, C. Courcoubetis, T. A. Henzinger, and P.-H. Ho. Hybrid automata: An algorith. approach to the specification and verification of hybrid systems. In *Hybrid Systems*, LNCS 1201, pages 209–229. Springer, 1993.

[AD94] R. Alur and D. Dill. A theory of timed automata. *Theoretical Comp. Science*, 126:183–235, 1994.

[CY98] P.W.H. Chung and S.H. Yang. Functional modeling for hazard identification. In *5th Int. Workshop on Functional Modelling of Compl. Technical Systems*, pages 117–124, Paris, France, 1998.

[DOTY96] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In T.A. Henzinger and E.D. Sontag, editors, *Hybrid Systems*, pages 208–219. LNCS 1066, Springer, 1996.

[EKKP95] S. Engell, S. Kowalewski, B. H. Krogh, and J. Preußig. C/E systems: a powerful paradigm for timed and untimed discrete models of techn. systems. In F. Breitenecker and I. Husinsky, editors, *EUROSIM 95*, pages 431–426, Wien, Austria, 1995. Elsevier.

[HLU+97] R. Huuck, Y. Lakhnech, L. Urbina, S. Engell, S. Kowalewski, and J. Preußig. Comparing TC/E Systems with Timed Automata. In *Int. Workshop on Hybrid and Real-Time Systems*, Grenoble, France, 1997.

[HTLW97] H.-M. Hanisch, J. Thieme, A. Lüder, and O. Wienhold. Modeling of PLC behavior by means of Timed Net Condition/Event Systems. In *IEEE Int. Symp. EFTA ´97*, pages 361–396, Los Angeles, USA, 1997.

[IEC93] IEC61131-3. *Programmable controllers, Part 3: Programming Languages.* IEC Geneva, Switzerland, 1993.

[KEPS99] S. Kowalewski, S. Engell, J. Preußig, and O. Stursberg. Verification of logic controllers for continuous plants using timed Condition/Event system models. *Automatica*, 35:508–518, 1999.

[Kle95] T. Kletz. *Comp. Control and Human Error.* Institute of Chemical Engineers, Rugby, UK, 1995.

[MPBC91] I. Moon, G.J. Powers, J.R. Burch, and E.M. Clarke. An automatic verification method using temporal logic for sequential chemical process control systems. *AIChE Journal*, 38, 1991.

[RYC98] G. Rong, S.H. Yang, and P.W.H. Chung. Hazard analysis for DCS based advanced control algorithm using functional model. In *Proc. Int. Conf. on Control*, pages 1090–1095, Swansea, UK, 1998.

[Stu00] O. Stursberg. Analysis of switched continuous systems based on discrete approximations. In *4th Int. Conf. on Automation of Mixed Processes: Hybrid Dynamical Systems*, Dortmund, Germany, 2000.

[TBK00] H. Treseler, N. Bauer, and S. Kowalewski. Model checking of control software under consideration of the PLC behaviour. In *5th Workshop on Discrete Event Systems*, Ghent, Belgium, 2000.