# Implementing a Visualization of an Industrial Production Cell Using Tk/Tcl[*] Technical Report

Artur Brauer, Claus Lewerentz, Thomas Lindner[†]

Forschungszentrum Informatik
Haid-und-Neu-Straße 10-14
76131 Karlsruhe
Germany
email: {brauer, lewerentz, lindner}@fzi.de

Oktober 26, 1993

Abstract

In this work, an application of Tk/Tcl to the domain of process visualization is described. We developed a simulation of an industrial production cell for to evaluate and validate control software for this (reactive) system. A major requirement was to provide a simple integration of the simulation with control programs.

In the first chapter the production cell and requirements for the visualization are described. Then the way the simulation was implemented using Tk/Tcl and our experiences are reported. This should be interesting for people who want to use Tk/Tcl for building visualizations of reactive or real time systems.

# 1. Problem

In the context of the German "Korso" Project (Correct Software) a number of formal software construction methodologies are developed. The Forschungszentrum Informatik (FZI) proposed the case study "production cell" to evaluate and compare these approaches. At FZI we built a realistic functional model of a small industrial production cell, sized about 1 x 1 meter. A schematic diagram of the cell is shown in figure 1.

The cell performs a production circuit, with metal blanks. Therefore, a blank is first put on the feed belt. The belt transports it to the elevating rotary table. The table rotates and lifts the blank to a position, where the robot can get it with arm 1. The robot transfers the blank to the press and releases it. When arm 1 has left the press, the press handles the blank. Arm 2 takes the blank outside of the press and puts it on the deposit belt. The deposit belt transports the blank to the robotal device, which discharges the deposit belt. The complete process isn't as simple as described above for several reasons. Since the robot have two arms it can handle two blanks at once, to improve the performance. What is more, the arms of the robot have different levels and can not move vertically. Such, the press have to balance the height difference of the arms. For a complete understanding of the model see [4].

* submitted to the Tk/Tcl Workshop 1993

† This work is sponsored by the German Ministry of Research and Technology (BMFT) as part of the compound project „KORSO — Korrekte Software".

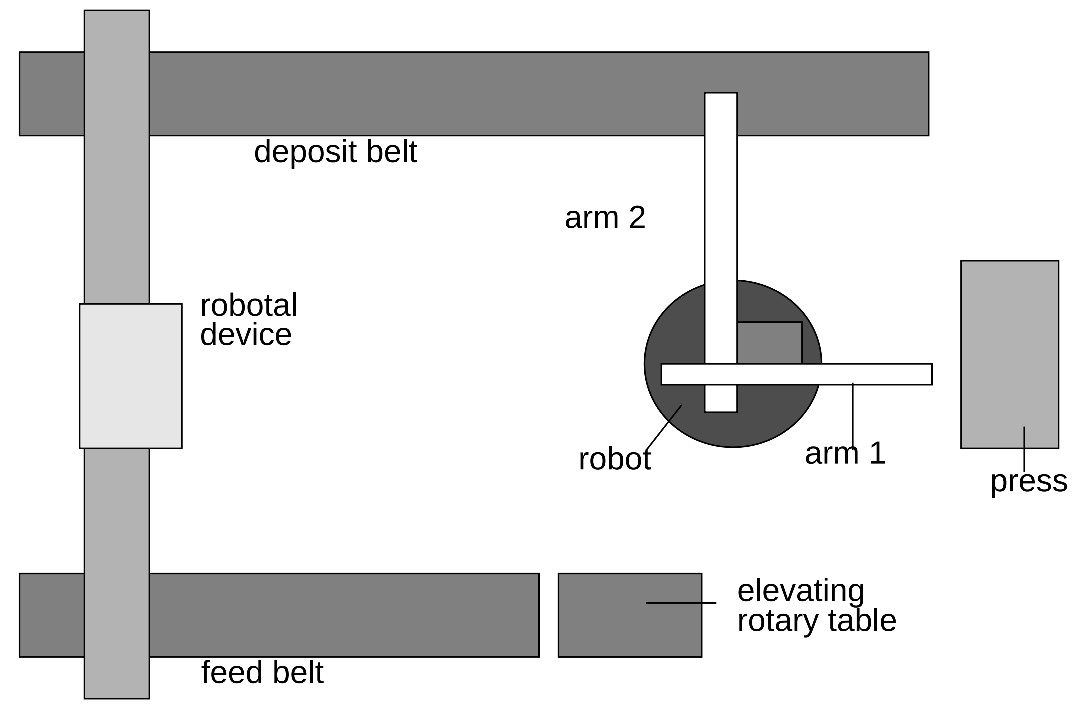

*Figure 1 Schematic diagram of the production cell*

The task of the case study, which is treated by several research groups in Germany, is the development of control software with various approaches, in order to compare the approaches and to show the usefulness of formal methods enforcing safety requirements in industrial applications. As the partners are distributed all over Germany it was necessary to provide them with a facility to evaluate their developments. We decided to construct a graphical software simulation of the production cell that should be easy to build and easy to connect with any control software.

## 2. Programming Language

For the implementation of the simulation we have chosen **Tk/Tcl**. Tk/Tcl consists of two parts. There is once Tcl, which is a simple interpretive command language, similar to Lisp. It provides variables, procedures and control structures like *if than else* and *for.* Also available are strings, lists and expressions. See [1] for a detailed description of Tcl. Tk is a toolkit for X11, which is implemented using Tcl, what means that it can be also used from Tcl applications. It provides various commands to create and configure widgets, to bind keystrokes and to deal with geometry managers.Especially, there are convenient commands to create and manipulate graphical widgets, called *canvas*. Furthermore, Tk provides a *send* command that allows any Tk-based application to invoke Tcl commands in any other Tk-based application. Using the send command applications can communicate in a powerful way. See [2] for a detailed description of Tk.

We decided to use Tk/Tcl by the following reasons: Firstly, Tk/Tcl enables one to program in X on a higher level. Especially the *canvas widget* helped us to speed up the application development. In previous experiments with Tk/Tcl we observed a much faster development compared with standard X programming. The second reason was that we had to provide an extremely simple coupling mechanism between the visualization and any control software developed by the project partners. A students work [3] had shown us that this coupling is very simple using Tcl mechanisms.

# 3. Experiences

Before and during the implementation of the simulation various decision had to be made. There was the whole structure of the simulation and the representation of the data to be determined. Also, many little details wanted to be resolved. This section describes considerations and decisions made and also the problems that occurred and methods that was taken to resolve them.

## 3.1 The common abilities s of the simulation

The simulation runs a window that contains a message widget and a graphical widget. Both widgets (and many more) are provided from Tk. Inside the graphical widget, which is called *canvas* in Tk, all the devices and blanks are drawn and than moved and rotated upon request. The message widget is used to print messages, which are mostly error messages of the devices. Figure 2 shows a screendump of the simulation together with a control panel, which runs as separate process.

Once the simulation is running its main task is to receive commands from a controller. These commands tell the simulation what the devices are supposed to do by setting flags. The simulation test the flags, computes the new positions of the devices and blanks and redisplays them. It also tests collisions and report them using the message widget in the top area of the window.

For more convenience a drag-and-drop mechanism for positioning the blanks was added. It allows to pick a blank by positioning the mouse pointer over it and pressing the left mouse button. The blank now can be moved to a device (just belts, positioning table and press) by holding down the button, moving the mouse pointer over the device and then releasing the button.

## 3.2 Helpful abilities of Tk/Tcl

Some abilities of Tk/Tcl were especially useful for the development of the simulation. Since Tcl is an interpreter language, there was no need to waste time on compiling source code. Moreover, using **wish**, a command line interpreter, allowed to test and debug the simulation cell's modules in a convenient way. A great help was e.g. the possibility to call procedures and change variables inside the simulation during the runtime.

Tk/Tcl provides additionally a set of commands to construct different kinds of widgets. One of the possible widgets is the canvas widget. It allows to draw lines, rectangles, ovals and so on. To each item can be assigned one or more tags. The items can be moved with the **canvas move** or the **canvas coords** command using the tags of the items to move. The command move moves one or more items just horizontal or vertical. The coords command draws an item at a new position, it's used by the simulation to rotate items, e.g. the robot.

In some cases it necessary to know which items are under a specified point or rectangle. This is e.g. when the robot wants to pickup a blank one have to know if the blank is under the magnet of the robot. Therefore, the **canvas find** command, which returns the indices of the items overlapping e.g. a rectangle, was used. A drawback of the find command is that it considers only rectangles that are parallel to the x- and y- axes.

## 3.3 The program structure

The simulation program has a very simple overall structure. This means that the simulation first creates all the widgets necessary and then builds up the scene and initialize all the variables. In our first release the simulation then entered the main loop from where it called the procedures to manage the blanks and update the devices, if this was required. The main loop was just left when the simulation was quitted.

First tests with a control software showed that a synchronous mode was useful. That means the simulation shouldn't do the next step on it's own. Instead, it should always wait on a react command to continue with the next cycle. This mode was added in the second release of the simulation. Therefor the main program tests after the initialization which mode is required and then either enters the main loop or stays in the event loop. (The event loop is entered automatically, when there are no more commands to be processed in a script.)

## 3.4 Performance Problems

Good performance was an important requirement, because the real time behavior of the production cell has to fit as closely as possible to the behavior of the hardware model. Early at the beginning of the development of the simulation it became obvious that moving several items would lead to a performance problem.

Tests showed that moving a raising quantity of items rapidly slowed down the simulation. In this context it is remarkable, that moving different items with several move or coords commands slowed down the simulation much more than a move command that was applied to several items, which all were connected with a common tag. Therefore, efforts were taken to use as little move and coords commands as possible. For the same reason very simple geometric shapes were used for the moveable items.

Another performance decreasing problem was calling procedures. Some tests with the procedure for rotating the robot were made. There was once a version that used procedures to compute transformed and rotated coordinates for the robot. This procedures were used very often. Another version computed the new coordinates itself. Since the second version was a little faster, it was chosen in spite of longer code.

A third method to improve the performance a little was to interleave the procedures for updating the devices, so just a smaller part of all update procedures is called at once in a cycle.

The last measure taken was to magnify the steps the devices were moved at once. A consequence was the lesser smooth movement.

## 3.5 Absence of composed datatypes

Sorrily, one is missing composed datatypes, like e.g. records in Pascal. The only datatypes supported by Tcl are string and array. We used lists to store all our composed data. Tcl provides very powerful commands to handle lists. A drawback that results from using lists is the aggravated oversight, since an element of a list it is indicated just by it's index, which doesn't say to much about it's content.

## 3.6 Aims for designing the update procedures

One of the main aims at the design of the device update procedures was to keep performance as well as possible (see also subsection 3.4 above). Moreover, the procedures were designed to perform all the action necessary for a device: moving the device, checking collisions occurred and also move the blanks that belong to the device. For the transitions of the blanks from one device to another a procedure was implemented that manages the blanks. It tests whether the conditions are satisfied to remove a blank from one device and assign it to another and performs all the actions that becomes necessary in this case.

## 3.7 Collision detection

The robotal device can collide with both belts. This occurs when the gripper is over the belt and then moves lower than the belt. Another possibility is when the gripper is lower than the belt and then moves over the belt. Both collisions are simple to prove, just by querying the position and the height of the gripper.

The robot and the press can clash, too. A collision occurs when the press's height turns into the region of the arm and the arm moves inside the area of the press. The height of the press is easy to query. For the test whether one of the arms of the robot is inside the press the canvas find command is used. It returns the items overlapping the press. Such, if an arm is among this items it is in the area of the press.

## 3.8 Raise and lower blanks and arm 2

To achieve a more realistic behavior of the simulation, in some cases it became necessary to raise and to lower items about or under other. When the press moves high enough then arm 2 of the robot can be move under the press without touching it. The arm and the blank that is maybe hold by the arm must be then drawn under the press. This is done using the **lower** command, which lowers an item under another specified item. The inverse case is when the press moves down low enough. In this case the lower command can also be used, since it draws an item just under a specified one and therefore even raises it. A similar case is when a blank is elevated by the robotal device from the deposition belt. It becomes necessary for the blank to be raised over all the other blanks since it will probably becomes the topmost of the blanks.The blank is lifted about all the other blanks using the command **raise**, which raises an item above a specified one.

## 3.9 Demonstration mode

After the simulation was finished, a demonstration mode, which could perform a demonstration of the simulation seemed to be useful. So a procedure was added, which starts and stops the devices such that a blank is handled through the production circuit. The procedure have several states, which it memorize in a global variable. Considering the current state it tests conditions for the devices and toggles them according to the result.

## 3.10 Coupling the Visualization and the Control Software

One of the main reasons to develop the simulation was that different control programs should be able to use it. Such an interface to the control software was needed. The **send** command, which was one of the main reasons to use Tk/Tcl, resolved this problem. The send command allows one Tk/Tcl application to send a command (note: a command and not just a string) to another

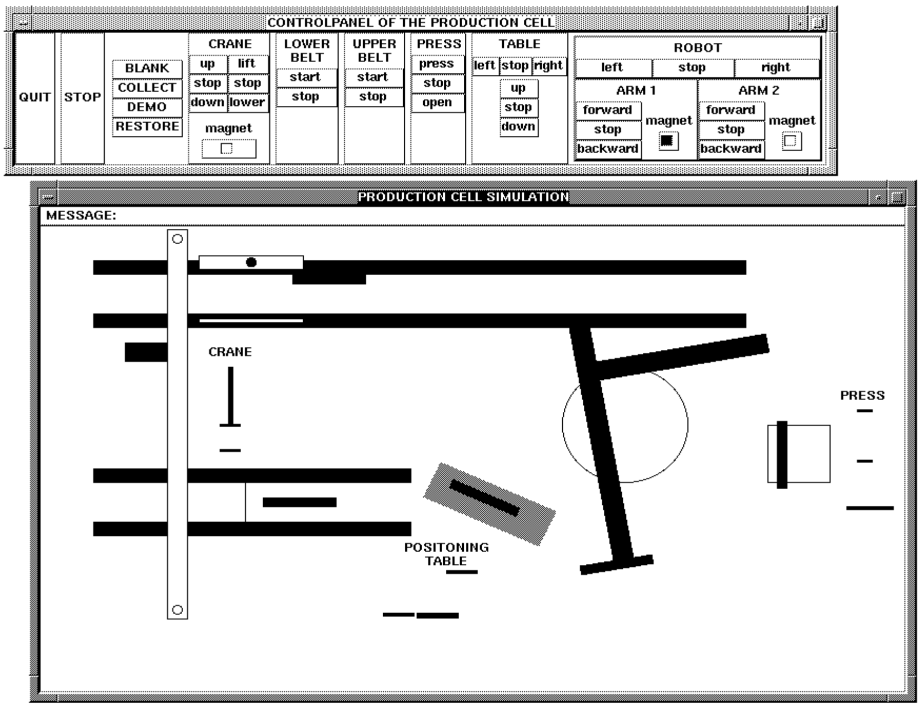

*Figure 2 Screen dump of the visualization (together with panel)*

Tk/Tcl application. The receiving application interrupts it's current operation and performs the received command. It is important to see, that the command can be a usual Tk/Tcl command but also a procedure that is known just to the receiver. Thus, the simulation need not to care active about receiving commands. It just contains a set of procedures, which change the states and flags of the devices. The simulation just tests the flags inside the main loop and call the device update procedures, if this is required. Since the control software should be free to use any language, a simple Tcl-script called *feedback_pipe* was added. Its task is to recognize commands issued from the control program (read in by stdin), and to call the respective procedures of the visualization using the send command. The very simple overall architecture of the application is shown in figure 3.

Additionally a simple ASCII communication protocol between the visualization and the control program (see section 5) was defined. A control program issues commands according to this protocol to *stdout*, and can thus switch the actors of the production cell on or off. With a special command also provided by the protocol definition, the control program can ask for the sensor status, which can then be read from *stdin*. The visualization was designed as a set of Tk/Tcl procedures, such that to each actor command there is a corresponding procedure. The status of the sensor values is recorded internally and, upon request, is written to *stdout*.

This coupling mechanism is flexible in two ways: of course, one can easily replace *prog1* by another control program, even by a panel for manual control (which was written using Tk/Tcl, too). Note that this would not be so simple if we had used UNIX named pipes. In addition, it is also possible to replace the simulation by a simpler Tcl-script, which directly sends commands and retrieves the state (via the serial port) from the hardware model. In order to achieve this, some small low-level C procedures were added to Tcl, thus resulting in a Tcl extension, that serves as a driver for the hardware model.

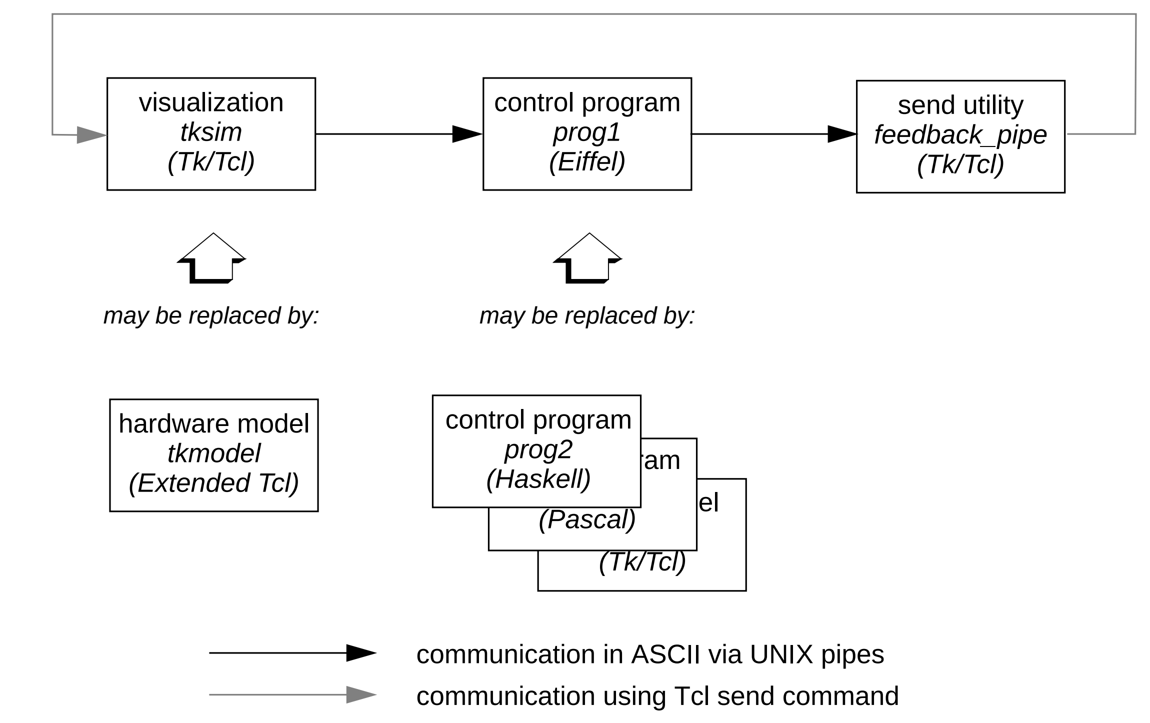

*Figure 3 Communication architecture*

## 3.11 Guards

We tested the simulation with a control software developed at the FZI by a student [5] and a problem became obvious. The control software reacted very slow, so it couldn't assure to stop a device when it had reached a desired position. Even more, in a bad case it couldn't stop a device to prevent it from a collision. So the simulation was augmented to provide guards. A guard is an object which consists of four elements: sensor number, operator, destination value and command. The sensor number identifies a sensor of a device (see also subsection 5.8), which provides information about the device. The guards are stored in a list called guardlist. In each passing of the main loop a procedure tests for each guard of guardlist whether the value of the sensor compared with the operator to the destination value gives a true expression. If the expression evaluates to true, the command is performed and the guard is deleted from the guardlist.

## 3.12 Starting the simulation

For a reliable cooperation between the feedback_pipe, the simulation and the control program the feedback_pipe of course have to know the name of the interpreter to which it have to send the commands. It also should be possible that the simulation can be started several times without affecting each other. For that reason a small program *startsimu* was written to start the simulation. It gives to the simulation and to the feedback_pipe a unique name using the process ID. Also, it acquaints this unique name of the simulation to the feedback_pipe. In this way the communication becomes sure.

To start the simulation you have to call **startsimu** from the directory that contains the simulation or from another directory giving the pathname to it. Startsimu will start the simulation and the feedbackpipe. It also will start a controller which is by default the control panel. To use your own controller you have to add the option **-con** followed by the pathname to your controller. If your controller requires the simulation to run in the synchronous mode you have to add the option **-snc** otherwise it will be started in the asynchronous mode. Other options possible are: **-bw** to use the monochrome mode and **-eng** and **-grm** to choose between the english and the german mode. As default the english language and color display will be used.

# 4. Implementation of the production cell simulation

This section describes in detail the implementation of the production cell simulation. Variables and procedures used are explained. Subsection 4.1 gives on overview of the structure of the implementation and in subsection 4.2 the scripts are explained.

## 4.1 Structure

The simulation consists of five parts:

- **tksim**, containing some setups and the main loop
- **siminit**.tcl, containing the graphics build-up procedures
- **simmotion.tcl**, containing the graphics movement procedures
- **simmath**.tcl, containing a few mathematical procedures
- **siminterface.tcl**, containing the control procedures, that correspond to the protocol

Also used by the simulation are the data files:

- **bitmap_grey.xbm**, which contains a bitmap that is used to fill the elevating rotary table, when monochrome mode is chosen
- **simcoltab_cl.tcl** and **simcoltab_bw.tcl**, containing the color values for color or monochrome display
- **simname_eng.tcl** and **simname_grm.tcl**, containing the english or german names for the devices used in the simulation
- **simerrors_eng.tcl** and **simerrors_grm.tcl,** containing the english or german error messages

The simulation was divided into several parts, since the overview suffered when it grew large. Also, it seemed useful to separate procedures with different kinds of functionality. The simulation should have been able to run on color and monochrome monitors. Therefore, the color description files **simcoltab_cl.tcl** and **simcoltab_bw.tcl** were introduced to describe the colors to use, depending on what display mode is chosen**.** A similar case resulted from the desired possibility to use english or german language. The files **simerrors_eng.tcl** and **simerror_grm.tcl** contain the possible messages (which are mostly error messages). The files **simnames_eng.tcl** and **simnames_grm.tcl** contain the names of the devices. Figure 4 shows the structure of the implementation.

## 4.2 Description of the Tk/Tcl Scripts

The simulation is divided into five parts to achieve a better overview. In the following subsection we describe in some detail the five Tk/Tcl-scripts out of which the simulation is composed.

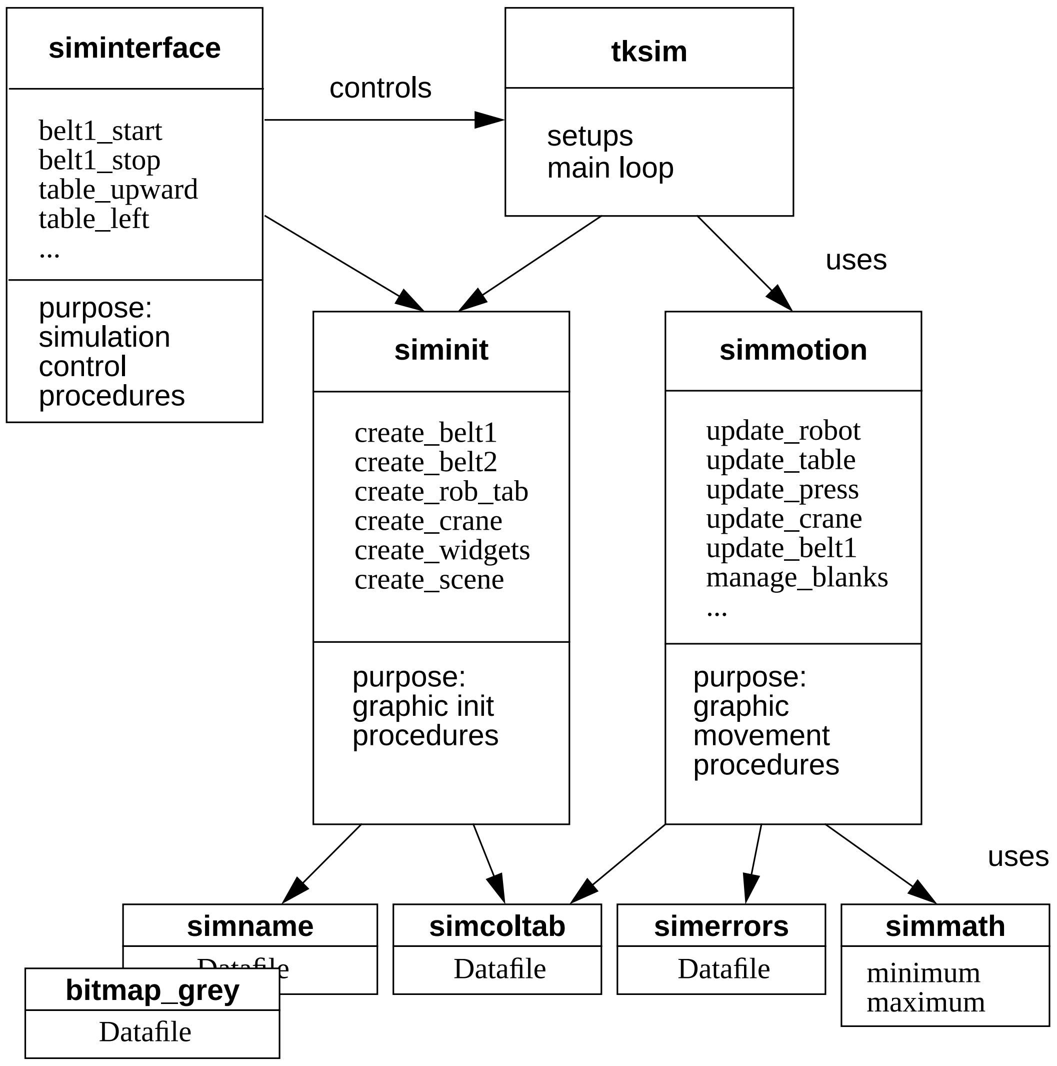

*Figure 4 Structure of the implementation of tksim*

### 4.2.1 Script *tksim*

This script contains the main program. Here the other scripts are included, setups are done and the widgets and graphics are build up, calling procedures of the other parts. After that, depending on the chosen mode the main loop or the event loop is entered and runs all the time unless the simulation is terminated. All the actions are described in detail below. In the first subsection we describe the scripts included, the second subsection gives a comprehensive survey of the cell's state described by some composed data structures. This survey is given in a tabular form. The last subsection comments on the main program.

#### 4.2.1.1 Include files

The script tksim includes all the other scripts, named in section 4.1, whereby a choice is made between color or monochrome display and between german or english language. This is done by evaluating the tcl-variable argv, which contains the parameters given when calling tksim. Possible Parameters are **-eng**, **-grm** and **-bw**.

**-eng** means that the language used by the simulation is english, which as also the default setting

**-grm** chooses the german language

**-bw** means that the display type is monochrome, otherwise colors are used

If the english language is chosen, the datafiles with the suffix *_eng.tcl* are included. For german output datafiles with the suffix *_grm.tcl* are included. For color output *simcoltab_cl.tcl* and for monochrome display *simcoltab_bw.tcl* is included. All the datafiles, except the bitmapfile, set global variables, which any procedure of the simulation can use.

#### 4.2.1.2 Data structures used in tksim

Most of the following description of the state information is done in a tabular form. Since Tcl supports only strings, all composed data types are lists.

##### 4.2.1.2.1 Variables to handle the cell

The most important variables to handle the simulation cell are **prod_b2**, **prod_b1**, **prod_a2**, **prod_a1**, **prod_tb**, **prod_pr**, **prod_cr**. For each device of the production cell one of this variables contains the whole information necessary to run it (e.g. the variable **prod_a1** contains the information for arm 1 of the robot). The information is stored as a list and is initialized at the creation of the corresponding device. The format of this lists is described below. The table's first column holds the index of the element in the list. The second column describes the kind of the information. Three types are possible:

- actor, which determines how the device has to be moved,
- sensor, which holds the device's state information, that can be read by an external controlling program,
- internal information, which is just used by the simulation.

The third column gives a short description of the element. The fourth column contains the possible values, if this is of importance. The last column describes the element more detailed.

**Format of the conveyor belts lists prod_b1 and prod_b2**

| Index | Actor/ Sensor | Description | Value/ Contents | Meaning |
|---|---|---|---|---|
| 0 | actor | movement | 0 | no movement |
| | | | 1 | movement left or right, depending on the belt |
| 1 | internal info | position of mark | | the position of the moveable mark on the belt |
| 2 | internal info | absolute length | | absolute length of the belt |
| 3 | internal info | blanks on the belt | | the list of the blanks that are on the belt |
| 4 | internal info | photoelectric barrier margin left | | the left margin of the photoelectric barrier on belt 2 |

| Format of the conveyor belts lists prod_b1 and prod_b2 | | | | |
|---|---|---|---|---|
| Index | Actor/ Sensor | Description | Value/ Contents | Meaning |
| 5 | internal info | photoelectric barrier margin right | | the right margin of the photo-electric barrier on belt 2 |
| 6 | sensor | photoelectric barrier | 0 | no blank inside |
| | | | 1 | blank inside |

| Format of the robot arms lists prod_a1 and prod_a2 | | | | |
|---|---|---|---|---|
| Index | Actor/ Sensor | Description | Value/ Contents | Meaning |
| 0 | actor | rotation | 0 | no rotation |
| | | | 1 | rotate left |
| | | | 2 | rotate right |
| 1 | actor | robot arm’s movement | 0 | no movement |
| | | | 1 | backward |
| | | | 2 | forward |
| 2 | actor | magnet | 0 | off |
| | | | 1 | on |
| 3 | sensor | rotation | 0..360 | degrees |
| 4 | sensor | extension | 0..absolute length | length of the extension of the arm, measured from the center of the robot to the magnet |
| 5 | internal info | absolute length | | total length of the arm |
| 6..9 | internal info | coordinates of the arm | four real numbers | the arm is drawn as a line deter-mined by two points |
| 10 | internal info | list of blanks | | names the blanks that are hold by the arm |
| 11 | internal info | maximum extension | | maximum possible extension of the arm |
| 12 | internal info | minimum extension | | minimum possible extension of the arm |

| Format of the elevating rotary table list prod_tb | | | | |
|---|---|---|---|---|
| Index | Actor/ Sensor | Description | Value/ Contents | Meaning |
| 0 | actor | rotation | 0 | no rotation |
| | | | 1 | left rotation |
| | | | 2 | right rotation |
| 1 | actor | up/down movement | 0 | no movement |
| | | | 1 | downward movement |
| | | | 2 | upward movement |
| 2 | sensor | rotation | 0..360 | angle of the elevating rotary table |
| 3 | sensor | height | | difference between top position and current position |
| 4..7 | internal info | coordinates of the table | four real numbers | the table is drawn as a line, determined by two points |
| 8 | internal info | blanks on the table | | list of the blanks that are on the table |
| 9 | internal info | possible height | | maximum possible height of the table |
| 10 | internal info | height of belt 2 | | the height of the deposit belt next to the table |

| Format of the press list prod_pr | | | | |
|---|---|---|---|---|
| Index | Actor/ Sensor | Description | Value/ Contents | Meaning |
| 0 | actor | movement | 0 | no movement |
| | | | 1 | downward movement (open) |
| | | | 2 | upward movement (close) |
| 1 | sensor | height | | the difference between top position and current position |
| 2 | internal info | blanks in the press | | list of blanks that are inside the press |
| 3 | internal info | absolute height | | maximum possible height of the press |
| 4 | internal info | height of arm 2 high | | height of arm 2 at the upper side |

| **Format of the press list prod_pr** | | | | |
|---|---|---|---|---|
| Index | Actor/ Sensor | Description | Value/ Contents | Meaning |
| 5 | internal info | height of arm 2 low | | height of arm 2 at the lower side |
| 6 | internal info | height of arm 1 | | height of arm 1 at the lower side |

| **Format of the crane list prod_cr** | | | | |
|---|---|---|---|---|
| Index | Actor/ Sensor | Description | Value/ Contents | Meaning |
| 0 | actor | movement | 0 | no movement |
| | | | 1 | movement toward belt 1 |
| | | | 2 | movement toward belt 2 |
| 1 | actor | lift /lower magnet | 0 | no vertical movement |
| | | | 1 | lower magnet |
| | | | 2 | lift magnet |
| 2 | internal info | position on the rail | | the position of the crane on the rail |
| 3 | actor | magnet | 0 | off |
| | | | 1 | on |
| 4 | sensor | height | | difference between top position and current position |
| 5 | internal info | maximum height | | the maximum possible height |
| 6 | internal info | height of belt 2 | | height of the deposit belt under the crane |
| 7 | internal info | height of belt 1 | | height of the feed belt under the crane |
| 8..11 | internal info | coordinates of the magnet | four real numbers | the crane’s arm is drawn as a line, determined by two points |
| 12..15 | internal info | coordinates of height mark | four real numbers | the crane’s height mark is drawn as a line, determined by two points |
| 16 | internal info | blanks hold by the crane | | list of blank that are hold by the magnet of the crane |
| 17 | internal info | stop at belt 2 | | the stop when moving the crane toward the deposit belt |

| Format of the crane list prod_cr | | | | |
|---|---|---|---|---|
| Index | Actor/ Sensor | Description | Value/ Contents | Meaning |
| 18 | internal info | stop at belt 1 | | the stop when moving the crane toward the feed belt |
| 19 | internal info | phot. bar. 2 margin left | | left margin of the photoelectric barrier over belt 2 (smaller y-value) |
| 20 | internal info | phot. bar. 2 margin right | | right margin of phot. bar. over belt 2 (greater y-value) |
| 21 | internal info | phot. bar. 1 margin left | | left margin of phot. bar. over belt 1 (smaller y-value) |
| 22 | internal info | phot. bar. 1 margin right | | right margin of phot. bar. over belt 1 (greater y-value) |

In addition to these variables, for each device exists a boolean variable, which is set to 1 if the device must be moved: **move_rob**, **move_b2**, **move_b1**, **move_tb**, **move_pr**, **move_cr**.

#### 4.2.1.2.2 Variables to handle the blanks

To manage the blanks there are following variables:

**max_blanks**, which determines how many blanks can be maximally used,

**loc_blanks**, which contains a list with an entry for each blank, that tells where the blank is. Therefore the first blank is assigned to the first element of the list, the second blank to the second element and so on. Each entry is a number between -1 and 7 and indicates one of the possible locations of the blank:

| | |
|---|---|
| -1 | blank has been dropped irregularly |
| 0 | blank is not in use |
| 1 | blank on first conveyor belt |
| 2 | blank on elevating rotary table |
| 3 | blank on arm 1 |
| 4 | blank in the press |
| 5 | blank on arm 2 |
| 6 | blank on second conveyor belt |
| 7 | blank on crane |

The boolean variable **add_blank** indicates when a blank is to be added to the production circuit.

#### 4.2.1.2.3 Other variables

Other variables used as global variables are:

**stop** this is a flag that indicates when the simulation is to be stopped

**errorlist** contains the error reports occurred since the last status report

**passings** counts the passings of the main loop, computed modulo 10000

**demo** a flag that is set when a the simulation shows a demonstration.

There is also a quantity of variables beginning with **color_**, **name_** and **errors_**. They are defined inside the scripts *simcoltab_*, *simname_* and *simerrors_*. The **color_...** variables contain color values, depending on the display mode chosen. The variables **name_...** hold the names of the devices and the variables **error_...** describes the error messages, all of them depend on the chosen language.

#### 4.2.1.3 Main program

The main program starts with building-up the widgets and graphics of the simulation by calling procedures of siminit. Hereafter the simulation branches either to the main loop (which represents the asynchronous mode) or enters the event loop (which represents the synchromous mode), corresponding to the required mode. To get into the main loop the procedure **enter_main_loop** is called. To enter the event loop no explicit command is necessary, it's automatically entered when no more commands exists to be performed.

The main loop is a cycle which calls the procedures to move or rotate the devices and manage the blanks, if the corresponding flags are set. Here some procedures are handled different. The ones to move the conveyor belts and the press, to add a blank, the management of guardslist and the management of the demonstration can be called in each passing of the loop. The procedures to manage the blanks, to rotate and move the robot and the table, to move the crane can be called only each third time. This method was chosen to interleave the graphical update procedures and improve a little the performance. The last command of the main loop is **update**. It is es especially remarkable, since it causes the interpreter to redraw the graphics and also to test wether a command was received from the feedbackpipe. In case a command was received the loop is interrupted and the command is executed.

Inside the event loop the simulation just waits on commands from the feedbackpipe. It doesn't perform updating the devices or anything else on it's own. To cause some motion, the user have to transmit the **react** command. The procedure react is similar to enter_main_loop except it doesn't consist of a cycle, so all device update procedures are called only once.

### 4.2.2 Script *siminit*

This script contains the graphics setup-procedures. To achieve a good performance all the moveable items are drawn as lines. This improves the performance and also simplifies their handling.

**- create_belt2 {canv length height xpos ypos}**

creates the deposit belt that moves to the left

**canv** name of the canvas to draw in the belt length

**length** of the belt

**height** of the belt

**xpox** the x-coordinate of the top left edge of the belt

**ypos** the y-coordinate of the top left edge of the belt

Both conveyor belts are build up of an underground rectangle, two margin rectangles and a line that represents the moveable mark on the belt.At the end of the belts is also a photoelectric barrier. Besides, at each end of both belts is an invisible rectangle. This provides that the mark disappears when it is moved outside of the range of the belt. In addition to the current version there were made tests with other kinds of conveyor belts. In one, the belt was build of several rectangles, that all have been moved. Another version built up the belt of many small lines drawn close to each other. A number of neighbored lines was drawn in the same color, then came some drawn in another color. This repeated permanently. The belt was moved e.g. right by changing the color of the first line of each color group. The current version was chosen because it seemed to have a better performance. Especially the second alternative was very slow.

TAGS CREATED:

The tag of the moveable mark of the deposit belt is ***blt2_tg***. The tag of the pad is ***blt2-pad_tg*** and the tag of the photoelectric barrier is ***blt2led_tg***.

**- create_belt1 {canv length height xpos ypos}**

creates the feed belt that moves to the right (see also create_band2 above)

**canv** the name of the canvas to draw in the belt

**length** the length of the belt

**height** the height of the belt

**xpos** the x-coordinate of the top left edge of the belt

**ypos** the y-coordinate of the top left edge of the belt

TAGS CREATED:

The tag of the moveable mark of the feed belt is ***blt1_tg*** and the tag of the pad is ***blt1-pad_tg***. The tag of the photoelectric barrier is ***blt1ed_tg***

**- create_rob_tab {canv xpos ypos l_a2 l_a1 l_tb h_tb}**

creates a robot, a elevating rotary table, a press and the blanks

**canv** name of the canvas to put the items in

**xpos** x-coordinate of the middle of the robot

**ypos** y-coordinate of the middle of the robot

**l_a2** length of arm 2

**l_a1** length of arm 1

**l_tb** length of the table

**h_tb** height of the table

Notice: when monochrome mode is chosen the table is created as stippled line, using the bitmap_grey.xbm bitmap file.

TAGS CREATED:

The tags of the blanks are ***blank0_tg***, ***blank1_tg***, and so on. The tag of the press is ***prs_tg***. To the press's height mark the tag ***prshgt_tg*** is assigned. The table and it's height mark have the tags ***tab_tg*** and ***tabhgt_tg*.** Arm 1 of the robot has the tag ***arm1_tg*** and arm 2 has the tag ***arm2_tg***. Additionally, ***allblnks_tg*** is assigned to all the blanks. It allows to raise a blank over all the other blanks.

**- create_crane {}**

creates the crane

TAGS CREATED:

The tag of the crane's moveable part is ***crn_tg*** and it's height mark has the tag ***crnhgt_tg***. The photoelectric barriers have the tags ***crnled1_tg*** and ***crnled2_tg***.

**- create_widgets {}**

creates all the widgets necessary for the simulation

**- create_scene{}**

creates the whole scene, is using the procedures create_belt1, create_belt2, create_rob_tab and create_crane

### 4.2.3 Script *simmotion*

This script contains all the procedures to animate the scene. Any motion of the items is achieved by using either the *canvas move* command or the *canvas coords* command. Both takes the tag of the items to move as parameter. Notice that the height of the crane, the press and the elevating rotary table is measured from the top position to the current position. Such height 0 means the element to be on top position (the press is closed, the elevating rotary table is just below arm 1, the crane is at the highest possible position). Raising height means that the item becomes lower. This method of measure was chosen because the top point of the items is determined by the model, whereby the bottom point is of no real interest.

**- update_robot {xpos ypos}**

turns and moves the robot around the point (xpos,ypos)

The procedure turns the robot left or right a constant angle and moves the arms forward or backward. It also tests if collisions occurred during the movement and reports them using melde. The angle to rotate the robot can be defined using the procedure choose_anlgle_rob (see below). The information whether the robot should be turned left or right and also whether the arms should be pushed is hold in prod_a1 and prod_a2. All the computations of the new coordinates are done inside this procedure. No other procedures are used for calculation. All the computations are also applied to the coordinates of the blanks, which are hold by the robot. The list of the blanks hold by arm 1 respectively arm 2 is stored in prod_a1 respectively prod_a2. Notice: at the moment the blank-management allows only one blank on one arm at the same time. After rotation and after pushing the arms collision detection is made. Collisions may only appear between robot arms and the press. If one of the arms is in the region of the press and the press is in the height of the arm then a collision is reported. The arms are recognized to be in the area of the press by using the canvas find command.

TAGS USED:

The procedure uses the tag of arm 1 ***arm1_tg*** and the tag of arm 2 ***arm2_tg***. Also used are the tags of the blanks hold by the robot.

**- update_table {xpos ypos}**

turns the elevating rotary table around the point (xpos,ypos) and actualize the table's height mark; detects collisions

If rotation is required the procedure computes the new coordinates of the table turning it an constant angle. The information about the rotation direction is got out of prod_tb. The rotation angle can be defined using the procedure choose_angle_tab. All the computations are also applied to the coordinates of the blanks, which are lying on the table. The list of the blanks is also hold in prod_tb. If movement up or down is required (prod_tb) then the height mark of the table is moved using the move command of Tk.

There is only one collision that may appear during moving the table: when the angle of the table becomes smaller then 0 degrees when moving it left. This is reported as collision with belt 1.

TAGS USED:

The procedure uses the tags of the elevating rotary table and of the height mark: ***tab_tg*** and ***tabhgt_tg***. Also used are the tags of the blanks on the table.

**- update_press {}**

moves the height mark of the press; detects collisions

This procedure moves the height mark of the press up or down, depending on what is required in prod_pr. Motion is done using the Tk command move. Collisions may appear, when an arm is in the region of the press and the press is opened or closed. For definition of *arm is in the region of the press* see description of update_robot above. Another action done in this procedure is to raise or lower arm 2 and perhaps the blank that is hold by it over or under the press. This becomes necessary when the press gets higher or lower then arm 2. For that reason the press is drawn over or under the arm 2 using the Tk command raise or lower.

TAGS USED:

The procedure uses the tags of the press, of the height mark of the press, of arm 2:***prs_tg***, ***prshgt_tg***, ***arm2_tg***. Also used are the tags of the blanks in the press.

**- update_crane {}**

moves the crane; detects collisions

This procedure moves the crane and the blank hold by it over the rail and actualizes it's height mark. It also detects collisions that occur during this movements.

Initially the movement of the crane over the rail is done, if this is required. (All the information concerning the crane is stored in prod_cr.) The crane and the blank is moved using the Tk move command. Therefore the blank gets added the tag crn_tg. When the crane moves into the region of one the photoelectric barriers, it's color is set to red, when the crane leaves the region the color is set back to green. Additional collision detection is done. A collision is reported when the crane moves into the area about a conveyor belt and the crane's height is greater than the height of the corresponding belt. In the second part of the procedure the height mark of the crane is moved up or down, if necessary, with the Tk coords command. Here collisions are detected, too. If the crane is over a conveyor belt and moves down his magnet, such that his height becomes greater than that of the belt, a collision is reported.

TAGS USED:

The procedure uses the tag of the part of the crane that is moved over the rail ***crn_tg*** and the tag of the height mark ***crn_tg***. To change the colors of the photoelectric barriers the tags ***crnled1_tg*** and ***crnled2_tg*** are used. Also used is the tag of the blank to move with the crane.

**- update_belt1 {}**

moves the feed belt and the blanks on it

The conveyor belt is driven by moving the mark on it right. Usually this is done using the move command, whereby the blanks on the belt get added the tag of the mark. If the mark is moved out of the range of the belt, it must be set back to the beginning of the belt. Therefore the belt's tag is deleted from the blanks and the mark is moved to the desired position with the move command. Than the blanks are moved right with the move command.

TAGS USED:

Used by the procedure is the tag of the mark of the feed belt ***blt1_tg*** and the tag of the blanks that are on the belt.

**- update_belt2 {}**

moves the deposit belt and the blanks on it

The conveyor belt is driven by moving the mark on it left. Usually this is done using the move command, whereby the blanks on the belt get added the tag of the belt. If the mark is moved out of the range of the belt, it must be set back to the beginning of the belt. Therefore the belt's tag is deleted from the blanks and the mark is moved to the desired position with the coords command. Than the blanks are moved left with the move command. The procedure also tests whether there is a blank inside the photoelectric barrier. Corresponding to the result of the test it changes the color of the photoelectric barrier.

TAGS USED:

The procedure uses the tag of the mark of the deposit belt ***blt2_tg*** and the tags of the blanks that are on the belt. The tag of the photoelectric barrier is ***blt2led_tg***.

**- manage_blanks{}**

manages all the blanks that are in use

For each blank the variable loc_blanks contains a numbers, which indicates the current location of the blank (see 3.1.2.2). Each device has also a variable that names all the blanks, that are to move with it. The management of this variables is done inside this procedure. Therefore it gets the current location of the blank from loc_blanks and proves the conditions to change to the next or to the previous location. If the proves have been successful the new location of the blank is entered into loc_blanks and the tag of the blank is deleted from the old device variable and added to the new device variable. Additionally other actions are possible, e.g. deleting the tag of the feed belt from the blank, if it has left the belt. The proves and actions are to numerous to be all explained here. Just one more should be described here. In case the blank lies on the deposit belt (this corresponds to location number 6) and all the conditions are true to pick the blank with the crane, then the blank is raised above the press and all the other blanks. For this reason all the blanks and the press have additional the tag prsblnks_tg.

TAGS USED:

The procedure uses the tag ***prsblnks_tg***, which allows to bring a blank in top position. Also in use are the tags of the blanks. In order to test whether a blank is over the feed belt or the deposit belt, the tags of the pad of belt 1 and belt 2 are utilized: ***blt1pad_tg*** and ***blt2pad_tg***.

**- put_blank{}**

puts a blank from the stock on the feed belt

TAGS USED:

The tags of the blanks.

**-return_dropped_blanks {}**

puts all the blanks that were dropped irregularly back to the stock

Each blank that was dropped irregularly, which is indicated by -1 in the loc_blanks list, is put back to it‘s place in the stock.

TAGS USED:

The tags of the blanks.

**- report_message {message number}**

reports error messages

This procedures writes the **message** to the message widget and adds the **number** of the error corresponding to the message to the list **errorlist**. The variable **errorlist** is used by the procedure get_status (siminterface).

**- choose_angle_rob {angle}**

determines the anlage to rotate the robot

This procedure determines the angle, which is used to rotate the robot. Possible angles are 1,2,3,4,5.

**- choose_angle_tab {angle}**

determines the angle to rotate the elevating rotary table

This procedure determines the angle, which is used to rotate the elevating rotary table. Possible angles are 1,2,3,4,5.

**- demonstration {}**

performs a demonstration of the simulation

**- manage_guards {}**

manages the guards and the guardslist

A guard consists of a **sensor_number**, an **operator** (one of: >=, <= and ==), a destination **value** and a **command**, which is a procedure from siminterface. A guard is stored in the list guardslist. The procedure tests wether the sensor with **sensor_num-**

**ber** compared with the operator to value is true or false. If the expression is true **ver-w_waechter** calls the **command** and removes the guard from guardslist. In this way it's possible to assure that a particular action is performed when a device reaches the desired position.

**- item_start_drag {canv x y}**

stores the coordinates of the blank inside the canvas canv, which overlaps the point (x,y) in the global variables lastX and lastY

The procedure tests wether a blank overlaps the point (x,y) and wether it is allowed to move it. A blank isn't allowed to move, if it is assigned to the crane or to the robot. If both conditions are satisfied the blank is removed from the current device using procedure blank_remove_from_dev and it's coordinates are stored in lastX and lastY.

**- item drag {canv x y}**

drags a blank inside the canves canv to the point (x,y)

**- blank_remove_from_dev { blank_number current_device }**

removes the blank with the number blank_number from the device idenified by current_device

**- blank_assign_to_dev { blank_number current_device }**

assigns the blank with the number blank_number to the device idenified by current_device

**- item_release {}**

release the current blank on the new device; uses blank_assign_to_dev

### 4.2.4 Script *siminterface*

This script contains all the procedures that are used to control the simulation. The procedures are all very short. They mainly just change flags.

**- arm1_forward {}**

starts the forward movement of arm 1

**- arm1_stop {}**

stops the movement of arm 1

**- arm1_backward{}**

starts the backward movement of arm 1

**- arm1_mag_on {}**

activates the magnet of arm 1

**- arm1_mag_off {}**

deactivates the magnet of arm 1

**- arm2_forward {}**

starts the forward movement of arm 2

**- arm2_stop {}**

stops the movement of arm 2

**- arm2_backward{}**

starts the backward movement of arm 2

**- arm2_mag_on {}**

activates the magnet of arm 2

**- arm2_mag_off {}**

deactivates the magnet of arm 2

**- robot_left {}**

starts the rotation to the left of the robot

**- robot_stop {}**

stops the rotation of the robot

**- robot_right {}**

starts the rotation to the right of the robot

**- table_left {}**

starts the rotation to the left of the elevating rotary table

**- table_stop_h {}**

stops the rotation of the elevating rotary table

**- table_right {}**

starts the rotation to the right of the elevating rotary table

**- table_upward {}**

starts the upward movement of the elevating rotary table

**- table_stop_v {}**

stops the upward or downward movement of the elevating rotary table

**- table_downward {}**

starts the downward movement of the elevating rotary table

**- crane_to_belt2 {}**

starts the crane moving toward the deposit belt

**- crane_stop_h {}**

stops the movement of the crane

**- crane_to_belt1 {}**

starts the crane moving toward the feed belt

**- crane_lift {}**

starts the crane's magnet moving upward

**- crane_stop_v {}**

stops the movement of the crane's magnet

**- crane_lower {}**

starts the crane’s magnet moving downward

**- crane_mag_on {}**

activates the crane’s magnet

**- crane_mag_off {}**

deactivates the crane’s magnet

**- press_upward {}**

starts the press moving upward

**- press_stop {}**

stops the movement of the press

**- press_downward {}**

starts the press moving downward

**- belt2_start {}**

starts the deposit belt to move left

**- belt2_stop {}**

stops the deposit belt

**- belt1_start {}**

starts the feed belt to move right

**- belt1_stop {}**

stops the feed belt

**- blank_add {}**

puts a blank from the stock on the feed belt, calls procedure teil

**- blanks_collect {}**

puts all the blanks dropped irregularly back to the stock, calls procedure einsammeln

**- system_demo {}**

starts a demonstration of the simulation

**- system_restore {}**

restores the simulation

**- system_stop {}**

stops any movement in the simulation

**- system_quit {}**

quits the simulation

**- do_get_status {}**

set the global status_flag to 1, so the procedure do_get_status will be called from the react or the enter_main_loop the next time

**- do_get_status {}**

writes the status information of the simulation to the standard output (see protocol)

**- get_status_intern {}**

returns the status information; the procedure is used by the simulation itself and doesn't write anything to stdout

**- get_passings {}**

set the global passings_flag to 1, so the procedure do_get_passings will be called from the react or the enter_main_loop the next time

**- do_get_passings {}**

writes the number of passings through the main loop to stdout, can be used to synchronize the control program with the simulation; the number of passings is computed modulo 10000

**- new_guard {sensor_number operator destination_value command}**

**sensor_number** a number of a sensor (see protocol)

**operator** an operator, possible are <=, >= and ==

**destination_value** value to reach by the sensor

**command** a valid simulation command, e.g. belt1_start

the procedure adds a new guard to the list guardslist (see also verw_waechter)

# 5. Protocol

The production cell simulation can be controlled by writing ASCII commands to *stdout* and reading status information from *stdin* (like described in subsection 3.10). The commands that can be used are described in this section. The format of the status information got back from the cell is also described here. If one compare the commands of the protocol to the procedures of the siminterface script, it will be obvious that the commands are just procedures of the simulation, that can be called from another Tk/Tcl program using the send command.

## 5.1 General commands

- **system_quit**

  quits the simulation

- **system_stop**

  stops any movement in the simulation

- **blank_add**

  puts a blank from the stock on the feed belt

- **blanks_collect**

  puts all the blanks dropped irregularly back to the stock

- **system_demo**

  starts a demonstration of the simulation

- **system_restore**
  restores the simulation

## 5.2 Commands to control the feed belt

- **belt1_start**
  starts the feed belt to move right
- **belt1_stop**
  stops the feed belt

## 5.3 Commands to control the deposit belt

- **belt2_start**
  starts the deposit belt to move left
- **belt2_stop**
  stops the deposit belt

## 5.4 Commands to control the elevating rotary table

- **table_left**
  starts the rotation to the left of the elevating rotary table
- **table_stop_h**
  stops the rotation of the elevating rotary table
- **table_right**
  starts the rotation to the right of the elevating rotary table
- **table_upward**
  starts the upward movement of the elevating rotary table
- **table_stop_v**
  stops the upward or downward movement of the elevating rotary table
- **table_downward**
  starts the downward movement of the elevating rotary table

## 5.5 Commands to control the robot

- **arm1_forward**
  starts the forward movement of arm 1
- **arm1_stop**
  stops the movement of arm 1
- **arm1_backward**
  starts the backward movement of arm 1
- **arm1_mag_on**

activates the magnet of arm 1

- **arm1_mag_off**
  deactivates the magnet of arm 1
- **arm2_forward**
  starts the forward movement of arm 2
- **arm2_stop**
  stops the movement of arm 2
- **arm2_backward**
  starts the backward movement of arm 2
- **arm2_mag_on**
  activates the magnet of arm 2
- **arm2_mag_off**
  deactivates the magnet of arm 2
- **robot_left**
  starts the rotation to the left of the robot
- **robot_stop**
  stops the rotation of the robot
- **robot_right**
  starts the rotation to the right of the robot

## 5.6 Commands to control the crane

- **crane_to_belt2**
  starts the crane moving toward the deposit belt
- **crane_stop_h**
  stops the movement of the crane
- **crane_to_belt1**
  starts the crane moving toward the feed belt
- **crane_lift**
  starts the crane’s magnet moving upward
- **crane_stop_v**
  stops the movement of the crane’s magnet
- **crane_lower**
  starts the crane’s magnet moving downward
- **crane_mag_on**
  activates the crane’s magnet
- **crane_mag_off**
  deactivates the crane’s magnet

## 5.7 Commands to control the press

- **press_upward**
  starts the press moving upward
- **press_stop**
  stops the movement of the press
- **press_downward**
  starts the press moving downward

## 5.8 Commands to get status information from the simulation

- **get_status**
  writes the status information of the simulation to the standard output

  This command causes the simulation to write all the status information to stdout, as a 15 element vector. Each element of the vector is separated by a newline. The status vector contains all the information that the real production cell can provide and additionally a list of errors that occurred since the last status report. The format of the status vector is described below.

| Index | Sensor corresponding to device | Description | Value/ Meaning | Notes |
|---|---|---|---|---|
| 1 | press | Press in bottom position | 1 = yes<br>0 = no | |
| 2 | press | Press in middle position | 1 = yes<br>0 = no | |
| 3 | press | Press in top position | 1 = yes<br>0 = no | |
| 4 | robot | Extension of arm 1 | 0..1 | Value 1 mens the arm is fully extended. |
| 5 | robot | Extension of arm 2 | 0..1 | Value 1 mens the arm is fully ex tended. |
| 6 | robot | angle of rotation of the robot | -100..70 | Value of 0 degrees means that the robot is in the starting position. |
| 7 | elevating rotary table | elevating rotary table in bottom position | 1 = yes<br>0 = no | |
| 8 | elevating rotary table | elevating rotary table in top position | 1 = yes<br>0 = no | |
| 9 | elevating rotary table | angle of rotation of the table | -5..90 | Value of 0 degrees means that the table is in the starting position. |
| 10 | crane | is the crane over the deposit belt | 1 = yes<br>0 = no | |
| 11 | crane | is the crane over the feed belt | 1 = yes<br>0 = no | |
| 12 | crane | height of the crane's magnet | 0..1 | Value 0 means the magnet is in top position. |

| Index | Sensor corre-sponding to device | Description | Value/ Meaning | Notes |
|---|---|---|---|---|
| 13 | feed belt | is a blank inside the photoelec-tric barrier | 1 = yes<br>0 = no | |
| 14 | deposit belt | is a blank inside the photoelec-tric barrier | 1 = yes<br>0 = no | |
| 15 | all | error report | | A list of errors, that occurred since the last status report. The list is enclosed in curly braces and each element is separated by a space. The errors are coded as shown in the next table. |

**The meaning of the error numbers**

| **Error-number** | **Meaning** |
|---|---|
| 0 | No error occurred; if this is true, no other elements are in the error list. |
| 1 | A blank dropped irregularly, during the passage from the feed belt to the elevating rotary table. |
| 2 | Collision between elevating rotary table and the feed belt. |
| 3 | elevating rotary table reached the right stop. |
| 4 | Robot reached the left stop. |
| 5 | Robot reached the right stop. |
| 6 | Arm 1 dropped a blank invalidly. |
| 7 | Collision between arm 1 and press. |
| 8 | Arm 2 dropped a blank invalidly. |
| 9 | Collision between arm 2 and press. |
| 10 | A blank dropped from the end of the deposit belt. |
| 11 | Collision between crane and the deposit belt. |
| 12 | Collision between crane and the feed belt. |
| 13 | Crane dropped a blank irregularly. |
| 14 | Crane reached the stop over the feed belt. |
| 15 | Crane reached the stop over the deposit belt. |

- **get_passings**

  writes the number of passings through the main loop to *stdout*

  This command causes the simulation to write the number of passings through the main loop to stdout. The number of passings is computed modulo 10000. This can be used to synchronize the control program with the simulation, since the performance of the simulation depends a lot on the host it is running on.

## 5.9 Advanced commands

**new_guard** ***sensor_number operator destination_value command***

a new guard is created

The simulation allows to create guards, which assures that a particular action is performed at the right time. A guard tests a condition until it becomes true and than call a simulation command and removes himself. The condition to be tested consists of a sensor, an operator and a destination value. The **sensor_number** describes which sensor should be compared with the **operator** to the **destination_value**. When this expression is true, **command** will be executed.

# 6. Conclusion

The use of Tk/Tcl allowed for building a quite complex animated graphical simulation in a rather short time. The entire effort including the learning of Tk/Tcl by one of the authors was about 120 hours. The Tk/Tcl interpreter proved to be a robust and reliable tool for this task. The smooth embedding in the standard communication mechanism of UNIX (using character streams and pipes) and the straight-forward extensibility of the Tcl interpreter by custom C functions were very useful for the integration of different parts of the system.

One major drawback of the current Tk/Tcl version in this application was the degradation of performance when relying extensively on procedures to structure the application. Additionally, powerful and suitable modularization and abstraction mechanisms for data structures as well as for functions would be very useful. We would like to have object-oriented concepts for structuring and reusing Tcl programs.

## References

[1] J. K. Ousterhout, An Embeddable Command Language, Proceedings of the1990 Winter USENIX Conference.

[2] J. K. Ousterhout, An X11 Toolkit Based on the Tcl Language, Proceedings of the 1991 Winter USENIX Conference.

[3] Kai Gutenkunst, Techniques for the coupling of user interfaces and applications, Forschungszentrum Informatik, Haid-und-Neu-Straße 10-14, D-7500 Karlsruhe 1, 1992 (in German language).

[4] Thomas Lindner, Case Study Production Cell: Task Definition, Technical Report, Forschungszentrum Informatik, Haid-und-Neu-Straße 10-14, D-7500 Karlsruhe 1.

[5] Marc Michaeli, Control Software for a Production Cell, Forschungszentrum Informatik, Haid-und-Neu-Straße 10-14, D-7500 Karlsruhe 1.

## Appendix: How to Install and Use the Visualization

Prerequisite: Have a running Tk/Tcl installation.

1. Get the file *visualization.tar.Z* from the FZI ftp server *gate.fzi.de* (Internet 141.21.4.3) in directory *pub/korso/production_cell/visualization/*. Use binary transfer mode.

2. Un-compress and un-tar the file. Check the completeness of the installation by comparing the contents of your current directory to the list of files in the README file.
3. Change the file *startsimu* by entering the correct path of your *wish* interpreter in the first line and set the variable *wishpath* in the third line also to the path for *wish*.
4. Under UNIX, now just enter *startsimu.* The simulation will come up, together with a control panel for hand steering, which was also implemented using Tk/Tcl. See subsection 3.12 for a detailed description of how to start the simulation.
5. Press the *DEMO* button in order to get a demonstration of a typical processing cycle of the production cell.

Please report any errors to *lindner@fzi.de*. Also comments are welcome.