Proceedings of 4th IEEE International Symposium on *High Assurance Systems Engineering*, Nov. 1999, Washington DC, USA

Formally Modeling a Metal Processing Plant and its Closed Loop Specifications

Enrico Tronci[1]
Dipartimento di Matematica Pura ed Applicata, Università di L'Aquila, Coppito 67100 L'Aquila, Italy
tronci@univaq.it, http://univaq.it/~tronci

Abstract

We present a case study on automatic synthesis of control software from formal specifications for an industrial automation control system. Our aim is to compare the effectiveness (i.e. design effort and controller quality) of automatic controller synthesis from closed loop formal specifications with that of manual controller design followed by automatic verification. The system to be controlled (plant) models a metal processing facility near Karlsruhe.

We succeeded in automatically generating C code implementing a (correct by construction) embedded controller for such a plant from closed loop formal specifications. Our experimental results show that for industrial automation control systems automatic synthesis is a viable and profitable (especially as far as design effort is concerned) alternative to manual design followed by automatic verification.

Key Words Formal Methods, Model Checking, Control Systems, Finite State Systems, Reactive Systems, Hybrid Systems, Discrete Event Systems, Supervisory Control, Manufacturing Systems.

1 Introduction

Industrial automation control systems are typical examples of safety critical embedded *control systems*. A control system can often be partitioned into two main subsystems: a *controller* and a *plant*. Roughly speaking: (in an endless loop) the controller observes the state of the plant and on the base of such state information sends commands (stimulus) to the plant to achieve predefined goals. The designer task is to devise a suitable controller for the given plant.

If we think about hardware design *the plant* corresponds to the datapath whereas *the controller* corresponds to the control logic. However for a control system designer the plant is given (by nature or by someone else) and cannot be modified. On the other hand usually in hardware design both the datapath and the control logic are to be designed. This situation typically leads to quite different design problems and approaches.

To use formal methods (namely: model checking) to verify our embedded controller (be it hardware or software) we need to write: formal specifications for the whole system (plant and controller), a finite state model of the plant, a program defining our controller (from which a finite state controller model is then obtained).

No matter how good our formal verification techniques are, testing and simulation will always be there. This means that, beyond defining our controller, a plant model will also be defined in order to run simulations. This even when formal methods are not used. Thus formal specifications seem to be the price to pay to use automatic formal methods. A question then comes naturally. From a plant model and formal specifications is it possible to automatically synthesize our embedded controller?

Using OBDDs [4] and model checking [8, 2] in [12, 13] it has been shown that when the plant can be modeled as a *Finite State System* (FSS) a (correct by construction) program (indeed a combinational circuit) implementing an embedded controller can be automatically synthesized from the given formal specifications for the closed loop system (plant + controller) and a plant (finite state) model. This saves on the coding effort for the controller (thus saving on design time and cost) and allows us to focus on formal specifications (thus improving design quality).

Note that often the word synthesis refers to the process of getting a low level (e.g. using netlists, Verilog, VHDL, C, etc) controller program from a high level description of the controller behaviour (e.g. [7]). Our starting point however consists of a (finite state) plant model and closed loop formal specs.

In this paper we present a case study applying the techniques developed in [12, 13]. Our goal is to automatically synthesize an embedded controller for the production cell described in [9] from the closed loop specifications also given in [9].

The problem of verifying a manually designed (using e.g. CSL, Esterel, Signal, Lustre, SDL) controller for this production cell has been studied with various approaches (including model checking) in [9]. Interactive deductive synthesis from formal specifications has also been studied in [5]. However, as far as we know, no automatic synthesis from formal specifications has been attempted so far.

We want to assess viability of automatic controller synthesis from formal specifications for industrial problems of reasonable size. The (thoroughly studied) production cell case study presented in [9] well serves our purposes, since we can compare our design effort and controller quality with those reported in [9]. Moreover we can run our controller using the freely available Tcl/Tk graphic simulator provided by FZI [14].

Automatic synthesis of *supervisory controllers* has

[1] This research has been partially supported by MURST funds.

been studied in, e.g., [1, 3, 10]. For industrial automation control systems a case study on automatic synthesis of *supervisory controllers* has been presented in e.g. [3]. However for such systems no case study about automatic synthesis of *controllers* from formal specifications has been previously presented. The results obtained from our case study can be summarized as follows.

- We succeeded in automatically synthesizing a controller for the given plant (production cell). This is a moderate size system with about 10^{12} states. Still, to the best of our knowledge, is the largest system for which automatic controller *synthesis* has been attempted so far. Note however that automatic *verification* via model checking of systems even larger than our is routine.
- It took us about 1 man-week to write formal closed loop specifications and plant model from the informal requirements in [9]. One more man-week was needed to write our interface to the FZI graphic simulator [14]. From our formal specifications and plant model we automatically get C code (correct by construction) implementing a controller for the production cell.

Our design time effort compares favourably with the time needed to carry out manual design followed by automatic verification (ranging from weeks to months in the papers in [9]). Manual deductive synthesis, although more general than our finite state approach, took about 2 months (and a high expertise) in [5].

- Our controller has C source code size 203K and executable code size 197K. This is acceptable in our case, although not particularly small. E.g. the controller executables obtained via manual design using Esterel or Signal have size, respectively, 46K and 90K [9].
- Our controller can handle up to 6 metal blanks simultaneously present in the production cell. This is the same or more than many of the manually designed controllers in [9] can do. Note however that, by suitably refining the requirements, the controller designed in [6] can handle up to 8 metal blanks in the system.
- We achieve a very high reusability. In fact automatic synthesis rests on the construction of a plant (finite state) model, which, in turn, is a suitable synchronous parallel of the processes modeling the production cell components (objects). We define generic models for various types of plant equipment and then instantiate such generic models to obtain finite state processes modeling the actual equipments for our particular plant.
- Automatic synthesis takes formal requirements and returns executable code. When, as it is often the case, requirements change many times during the design, our approach improves the whole design process by relieving the designer from modifying the control software again and again and by allowing him/her to concentrate on writing (hopefully) correct (formal) requirements.

Incidentally we also found 3 requirements that are missing in the list of requirements given in [9]. These missing requirements are not too important when doing manual design since they are "obliviously" added by the designer using his intuition. However such missing requirements are essential when doing automatic synthesis. Indeed, in our experience, automatic synthesis from formal specifications coupled with (traditional) simulation is an effective way of detecting missing or wrong requirements since no programmer intuition *rescues* us from bugs in requirements.

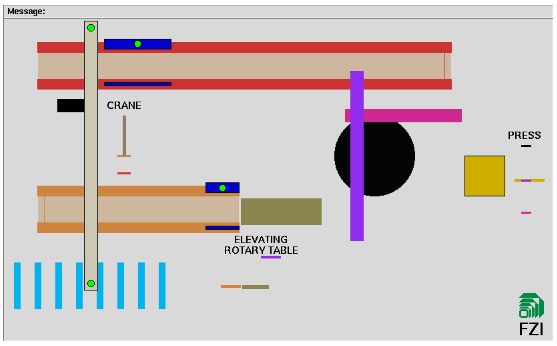

Figure 1: Production Cell (Tcl/Tk graphic simulator from FZI)

2 Production Cell

The production cell [9] is composed of two conveyor belts, an elevating rotary table, a two-armed robot, a press and a traveling crane. Figure 1 shows a top view of the production cell as it appears in the graphic simulator [14]. On the bottom [top] left is the feed [deposit] belt.

In order to perform demonstrations with the model, the production sequence has been made cyclical by using a traveling crane to move "forged" metal plates (which the press in the model does not actually modify) from the deposit belt back to the feed belt.

For validation purposes the organizer of this case study [9] have implemented a Fischer-Technik toy model controllable via an RS-232 serial line port as well as a Tcl/Tk graphic simulation environment. Both can be controlled using the same ASCII protocol. Thus the same control program can be used to operate the (hardware) toy model as well as the graphic simulator. For simulation purposes we used the graphic simulator which is freely available in [14].

The control program drives the plant by sending commands to the actuators and gets information about the status of the plant through sensors. E.g. some of the commands and sensors available are given respectively in figs. 2, 3. Note that the list of all sensors also defines the (observable) state components of the plant.

3 Production Cell Formal Model

The first step in automatic controller synthesis is to build a formal model of the system to be controlled (usually called *plant*). For us the plant is a *Finite State System* (FSS) consisting of all possible *safe* transitions that the controller can trigger (by sending commands to the plant).

Thus we can obtain a plant model by removing from our model of the production cell (*physical plant*) all *un-*

Command	Short Description
Elevating rotary table	
table_left	Starts rotation to the left of the elevating rotary table
table_stop_h	Stops rotation of the elevating rotary table
table_right	Starts rotation to the right of the elevating rotary table
table_upward	Starts upward movement of the elevating rotary table
table_stop_v	Stops vertical movement of the elevating rotary table
table_downward	Starts downward movement of the elevating rotary table
Feed belt	
belt1_start	Starts the feed belt to move right
belt1_stop	Stops the feed belt

Figure 2: Plant Commands

Device	Short Description	Range or meaning
Elevating rotary table (ERT)	Is the ERT in the bottom position?	1=y/0=n
	Is the ERT in the top position?	1=y/0=n
	Angle of rotation of the ERT	$-5..90$
Feed belt	Is a blank inside the photoelectric barrier?	1=y/0=n

Figure 3: The status vector

A	-5	(-5, 0)	0	(0, 50)	50	(50, 85)	85	(85, 90)	90
D	n5	n5p0	p0	p0p50	p50	p50p85	p85	p85p90	p90

Figure 4: Table angle AD conversion

```
tabh [tabv] = table horizontal [vertical] postion;
tabl = table load (by blanket);
arm1 [arm2] = position robot arm 1 [2];
mag1 [mag2] = robot magnet of arm 1 [2];
crane_mag = crane magnet;
deposit = deposit belt;
deposit_blank = deposit belt photoelectric cell;
craneh [cranev] = crane horizontal [vertical] position;
crane_mag = crane magnet;
nop is an event leaving state unchanged;
robot state values are prefixed with rb_;
tabl_px, tabl_nx, tabl_u: present state, next state,
  event variables for process table_load.
```

Figure 5: Linking ids to informal requirements

safe transitions. In this section we give an outline of how we model our production cell using FSSs.

We model each cell component as an FSS. An FSS, in turn, is represented with its transition relation. The physical plant model is obtained by building a suitable synchronous parallel of the FSSs modeling the cell components.

Each of the cell components is indeed built out of more elementary FSSs modeling some specific dynamic behaviour. Such elementary FSSs in turn are obtained by instantiating generic parametric models. This is important to increase *reusability* of specifications as well as for *correctness*. In fact correctness of the automatically synthesized controller rests on the correctness of our automatic synthesis program (akin to a compiler) as well as on the correctness of our plant formal model. Thus it is important that our plant model be simple and trustworthy.

Modeling Language

We use *First Order Logic* on a *Boolean* domain (BFOL) as our formal language. This allows us to easily define transition relations as well as safety conditions and to use OBDDs [4] to carry out our computations. This is by no means the only possible choice, but in our case it is a natural and easy one.

Essentially BFOL can be seen as (formal) syntactic sugar for C based OBDD [4] programming. Our interpreter from BFOL to OBDD is called BSP (*Boolean Symbolic Programming*) [11]. It has a LISP like syntax that we will only show by examples.

We define a process (i.e. an FSS) with its transition relation. Thus for each process p we have a boolean function (also named p) defining the transition relation of process p. We will denote with p`_px`, p`_nx`, p`_u` the arrays of boolean variables ranging, respectively, on p present states, p next states, p events (actions). E.g. variable `tabh_px` ranges on present states of process `tabh` modeling the horizontal motion of the rotary table.

For clarity in our BSP programs we prefixed state values for process p with p_, however we will drop such prefixes in our informal presentation since no ambiguity arises from doing so.

In the following we use pictures instead of BSP code to define processes. We use BSP code to define logical operations. The BSP term `(def` p e`)` defines the boolean function p to be e. The expression `(eqv` a b`)` is true iff the boolean arrays a and b are bitwise equal.

Controller Memory

To model our cell and to synchronize cell components we need some more information w.r.t. that provided by the available cell sensors. E.g. we need to know if at a certain time there is a blank on the feed belt or not (even when such blank is not under the photoelectric cell). This information is not directly available from the cell sensors, however it can be reconstructed from the sequence of values taken by the cell sensors. E.g. we have in our plant model a book-keeping process `feedbelt_load` (fig. 6) telling us if there is or not a blank on the feed belt.

Processes like `feedbelt_load` are indeed part of the controller. However, since our automatically synthesized controller ([12], [13]) is memoryless (i.e. it can be implemented as a combinational circuit), for synthesis purposes we consider processes like `feedbelt_load` as part of our plant. Once we have a memoryless controller for our plant we can get a controller for the real plant by simply moving all processes like `feedbelt_load` from the plant to the controller which will then be a (deterministic) sequential machine. This is the approach that we follow here.

As a side effect this approach clarifies also which hypotheses are needed on the controller initial state when the all system (plant + controller) is started (or

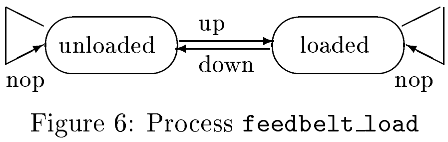

Figure 6: Process `feedbelt_load`

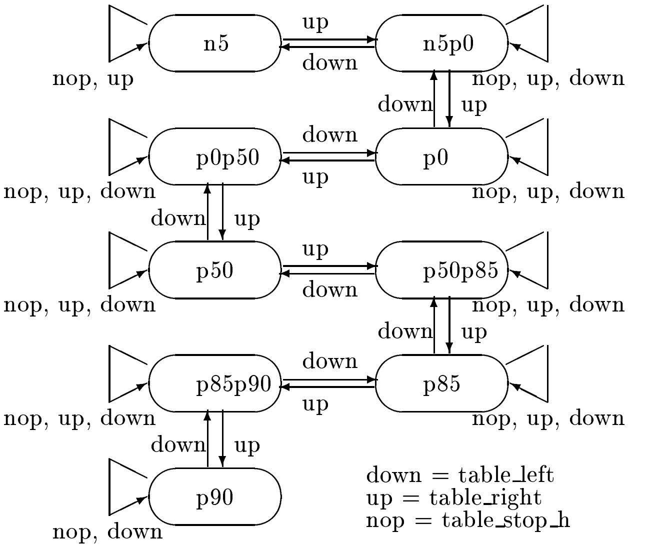

Figure 7: Process `tabh`

restarted after a failure). E.g. when we start the all system we need that the state of process `feedbelt_load` is `unloaded` if no blank is on the feed belt and `loaded` otherwise. Only if this holds the controller will be able to correctly compute the feed belt loading state using the available sensor information.

Phase Functions

Each cell component works accordingly to a fixed sequence of operations (phases). This behaviour is given in the requirements in [9] thus it will be part of our plant model. For process p this is modeled by function *p*`_phaser`. E.g. for the rotary table phasing is defined by function `table_phaser`.

As a result the controller task essentially is to decide (based on sensor information) for each cell component when to switch from one phase to the next one. This must be done optimizing performance without violating the given safety requirements.

AD conversion

Potentiometers are used to measure displacements (robot arms) as well as angles (table, robot). Potentiometers return continuous values. However only a few milestone values are relevant for the controller. Analog to Digital (AD) conversion is done taking this into account and this is why we can model the production cell using finite state automata. Our AD conversion for the table angle is shown in figure 4.

Note that there is no Digital to Analog (DA) conversion since we are only sending discrete commands to the plant (see fig. 2).

Table

The elevating rotary table is characterized by two values, namely its height and its rotation angle. These values can be modified by sending commands to the motors driving the table. Thus the elevating rotary table is described by two processes: `tabh` and `tabv` modeling respectively the horizontal and vertical motion of the table.

We show how we model the table horizontal motion. The table vertical motion, the robot, the press and the crane subsystems can all be modeled following (indeed suitably reusing) the model developed for the table horizontal motion (`tabh`).

We are not interested in all possible values of the table angle. We only need a few *milestones* as shown in fig. 4. In fig. 7 is the definition for process `tabh`.

At first sight the nondeterministic transitions in `tabh` may appear strange. However we must keep in mind that we are approximating with finite states (milestones) a continuous system. Thus each state in `tabh` represents in fact an interval of continuous values in the physical system.

The continuous dynamics for the table horizontal motion can be modeled with the equation $x(t+1) = x(t) + \alpha \cdot T \cdot u(t)$, where $t \in \mathbf{Nat}$ is the discrete time, $x(t)$ is the table horizontal position ranging on the interval [-5, 90], T is the sampling time, α is the motor speed, $u(t) = -1, 0, 1$ is the control input for the motor with -1, 0, 1 representing, respectively, `down`, `nop`, `up` in fig. 7. Of course this is a deterministic dynamics. However our digital controller will only be able to observe a finite set of values for the table position $x(t)$. Namely, when the (true) table position is $x(t)$ our digital controller will observe $\mathrm{AD}(x(t))$ where the function AD is essentially given in fig. 4. For each value of $x(t)$ our digital controller $\mathcal{K}$ will have to decide which command (among -1, 0, 1) it will send to the motor. To make this decision $\mathcal{K}$ needs to predict the effect of each command. Because of quantization $\mathcal{K}$ only knows that when the table position is $\mathrm{AD}(x(t))$ issuing command $u(t)$ will move the table to $\mathrm{AD}(x(t+1))$.

Let $\alpha = 1$, $T = 0.1$ and $u(t) = 1$. Consider the following two scenarios.
1. $x(t) = -1$. Then: $x(t+1) = -1 + 1 \cdot 0.1 \cdot 1 = -0.9$, $\mathrm{AD}(x(t)) =$ `n5p0`, $\mathrm{AD}(x(t+1)) =$ `n5p0`.
2. $x(t) = -0.1$. Then: $x(t+1) = -0.1 + 1 \cdot 0.1 \cdot 1 = 0$, $\mathrm{AD}(x(t)) =$ `n5p0`, $\mathrm{AD}(x(t+1)) =$ `p0`.

Thus, because of quantization, what the controller $\mathcal{K}$ *sees* is that starting from the same state `n5p0` and with the same command ($u(t) = 1$, i.e. `up`) the resulting table position may be `n5p0` as well as `p0`. Indeed one may argue that also the state `p0p50` may be reached in this situation, e.g. starting from $x(t) = -0.05$. However the 0 in the row (0, `p0`) in fig. 4 is in the actual AD converter a small interval I_0 centered around 0. Because our sampling time is small compared to motor speed we always go through I_0 (i.e. `p0`) when going from `n5p0` to `p0p50`.

Thus the dynamics for the table horizontal motion, *as seen by the digital controller*, is a nondeterministic process. This is just a consequence of quantization. Note however that our controller $\mathcal{K}$ will be deterministic, being it a function from states to commands. Of course the same considerations apply to the table vertical motion, the robot rotation, the motion of robot arms, the press motion, and the crane horizontal and vertical motions.

This nondeterminism in the physical plant model *as*

```
/* Sync between feedbelt and table */
(def sync_ft (and
(eq (and  (eqv feedbelt_sensor_px blank_in)
          (eqv tabh_px tabh_p0) (eqv tabv_px tabv_bottom)
          (eqv tabl_px unloaded) (eqv feedbelt_u go) )
    (eqv tabl_u begin_loading) )
(eq (and  (eqv feedbelt_sensor_px blank_in)
          (eqv tabh_px tabh_p0) (eqv tabv_px tabv_bottom)
          (eqv tabl_px loading) (eqv feedbelt_u go)  )
     (eqv tabl_u keep_loading) )
(eq  (and (eqv feedbelt_sensor_px blank_out)
          (eqv tabh_px tabh_p0) (eqv tabv_px tabv_bottom)
          (eqv tabl_px loading) )
     (eqv tabl_u end_loading) ) ... )
```

Figure 8: Feedbelt-Table synchronization

```
(def safe_x_1 (imp
   (and (eqv tabl_px loaded) (eqv arm1_load_px arm1_loaded))
   (or (leq robot_px rb_p0) (eqv arm1_px arm1_retracted)
       (eqv tabv_px tabv_bottom)) ))
(def safe_xu_6
   (imp(grt robot_px rb_p50)(eqv robot_u down)))
(def safe_xu_7 (imp (eqv robot_px rb_p50)
       (or (eqv robot_u down) (eqv robot_u nop))))
(def safe_xu_11
   (imp (and (eqv mag1_px mag1_off) (eqv mag1_u on))
       (geq robot_px rb_p0)))
(def safe_x (and safe_x_1 ... safe_x_6 ))
(def safe_xu (and safe_xu_1 ...... safe_xu_20 ...))
```

Figure 9: Safe states and transitions

seen by the digital controller cannot be removed since by doing so the controller makes wrong predictions on the plant behavior. This may result in the controller driving the plant to an unsafe state. E.g. removing the nondeterminism in our robot modeling will lead the controller to believe that in one sampling interval it can open the press and simultaneously retract a robot arm. This is indeed not the case because of motor speeds. The result is a crash between a robot arm and the press.

Note that using an event based plant modeling (rather than our sampling modeling) leads to the same situation since events must be triggered by (sensor) measures which, in turn, will be done by periodically sampling the plant.

Nondeterminism in the plant model may pose serious problems for controller synthesis because the outcome of a command is not known. E.g. suppose that process `tabh` is in state `n5p0` and that the controller issues command `up`. Because of nondeterminism, we can only say that `tabh` *may* reach state `p0` (but `tabh` may stay in state `n5p0` as well). However in our case we do know that if the controller issues command `up` from state `n5p0` then *eventually* process `tabh` will reach state `p0`, although we do not know how long this will take. This means that any controller that *may* drive our plant to a goal state *will* indeed do so. In turn this means that we can use the controller synthesis algorithm in [13].

Synchronization

Plant components exchange blanks. E.g. the rotary table gets a blank from the feed belt, the robot gets a blank from the rotary table as well as from the press, the press and the deposit belt get blanks from the robot, the crane gets a blank from the deposit belt, the feed belt gets a blank from the crane. Blank exchange can be modeled as process communication via suitable synchronization constraints saying when blank exchange is possible and what happens upon a blank exchange.

In fig. 8 we show part of the function modeling synchronization between the feed belt and the rotary table. All other synchronizations can be modeled along the same lines.

Function `sync_ft` in fig. 8 depends on present states and events and says which transitions are allowed in a certain plant state. E.g. from fig. 8 we see that if there is a blank under the feed belt sensor and the table horizontal position is 0 and the table vertical position is bottom and there is no blank on the table `(eqv tabl_px unloaded)` then the feed belt can move `(feedbelt_u go)` iff the process `table_load` begins loading (thus recording that a blanket is moving from the feed belt to the table).

```
(def sync (and sync_ft sync_tr sync_pr sync_dc))
(def unsafe_plant (and feedbelt feedbelt_load table_load
     magnet_1 magnet_2 arm1_load arm2_load press_load
     press_forge deposit deposit_blank deposit_load
     crane_load crane_mag tabv tabh arm1 arm2 robot press
     crane_hpos crane_vpos))
(def safex2xu_all (forall plant_all_nx (imp unsafe_plant
     (compose safe_x plant_all_px plant_all_nx))))
(def plant_all
     (and unsafe_plant safe_xu safex2xu_all sync))
(def end_all (and feedbelt_phaser table_phaser
     robot_phaser press_phaser deposit_phaser
     crane_phaser))
```

Figure 10: Plant and final states

Final States

The set of final (goal) states is the set of states that the controller strives to reach. It is defined with its characteristic function (`end_all` in fig. 10) which is simply the logical *and* of the phase functions (sec. 3).

The synchronization (`sync` in fig. 10) between cell components is such that when a cell component reaches a certain phase a transition is triggered making the phase function for that cell component true only on states belonging to the next phase. As a result the controller will drive each cell component endlessly from one phase to the next one.

E.g. the phaser function for the table roughly says: the controller goal is to load the table when it is unloaded and to unload the table when it is loaded. Now suppose that the table is unloaded. Then the controller will try to load it. As soon as the controller succeeds in loading the table the controller goal becomes that of unloading the table. As soon as the controller succeeds in unloading the table the controller goal becomes that of loading the table, and so on. This will keep the table moving for ever from one phase to the next one.

Our phase functions (sec. 3) essentially state that, for each cell component, once a blank is received it will be processed accordingly to the given requirements and then delivered to the next cell component. Thus the logical *and* of the phase functions in `end_all` (fig. 10) indeed formalizes the liveness requirement given in [9], i.e.: "every blank introduced into the system via the feed belt will eventually be dropped by the crane on the feed belt again and will have been forged".

Plant state bits	Plant state space size	CPU time (min)	Max OBDD	Ctr MUX	Ctr C code size (bytes)	Ctr exe size (bytes)	Ctr min rct t (ms)	Ctr max rct t (ms)	Ctr avg rct t (ms)
42	$4.4*10^{12}$	15:14	1,851,657	4852	202,660	197,248	0.0	10.0	0.6

Figure 11: Experimental Results on Linux PC with 200 MHz Pentium and 112MB of RAM

4 Safety Requirements

The control program must meet various safety requirements. Each safety requirement is a consequence of one of the following principles: restrict machine mobility, avoid machine collisions, do not drop metal blanks outside safe areas, keep blanks sufficiently distant.

In this section we list some of the given informal safety requirements (a complete list is in [9]) and give our formal version. We wrote our formal requirements by closely following the informal requirements given in [9]. This is easily possible since we are using first order logic (on booleans) and we have both action (event) and state variables.

We divide our formal safety requirements into two groups. Requirements stating when a state is safe (named `safe_x_...` in fig. 9) and requirements stating when a transition is safe (named `safe_xu_...` in fig. 9). We will see the reason for this division in sec. 6. Function `safe_xu` [`safe_x`] in fig. 9 defines the set of transitions [states] that are considered safe in the requirements given in [9].

All numeric constants we use in our formalization are given in [9].

Restrict Machine Mobility. The electric motors associated with some actuators (robot, arms, press, crane) may not be used to move the corresponding device further than necessary. E.g. the robot must not be rotated clockwise if arm1 points towards the elevating rotary table (`safe_xu_6, safe_xu_7`), and it must not be rotated counterclockwise if arm 1 points towards the press.

Avoid Machine Collision. To avoid machine collisions some requirements must be met. E.g. the rotary elevating table and the robot can collide if both arm 1 and the table hold a blank and the table is in top position. In this case the two blanks will collide, what is reported as a collision. To prevent this, one of the following conditions must be satisfied: the angle of the robot is less than or equal to 0, the extension of robot arm 1 is 0, the table is in bottom position. This requirement is formalized in `safe_x_1`.

5 Missing Requirements

By running the simulation with the controller obtained from our first formalization attempt we found that our controller was not working. Namely: blanks were never dropped since the magnets where kept off for too short a time interval. After turning off a magnet holding a blank the controller assumes that this blank is dropped since no sensor is available to check this fact. When, as in our case, this is not true, the plant is quickly driven to an unsafe state. Adding the following requirements fixes the problem.

1. Do not turn on robot arm 1 magnet too soon after having turned it off (`safe_xu_11`, fig. 9);
2. Do not turn on robot arm 2 magnet too soon after having turned it off;
3. Do not turn on the crane magnet too soon after having turned it off.

As far as we can tell the above requirements are not in the list given in [9]. When designing by hand this is not a big problem since the above requirements are more or less unconsciously added by the designer. On the other hand the above requirements must be explicitly stated in our formal model when doing automatic synthesis otherwise the automatically synthesized controller will not work as expected.

Indeed we found that automatic synthesis from formal requirements coupled with simulation is an effective way to debug requirements.

6 Plant Model

The transition relation for the plant process `plant_all` is given in fig. 10, where `unsafe_plant` is the transition relation for our production cell.

We need to restrict (logical *and*) `unsafe_plant` behaviour to transitions that are safe (defined with `safe_xu` in fig. 9) and that may only lead to a safe state (defined with `safex2xu_all` in fig. 10).

In function `safex2xu_all` in fig. 10 `(compose safe_x plant_all_px plant_all_nx)` is obtained from `safe_x` by simultaneously replacing `plant_all_px` with `plant_all_nx`.

From a theoretical point of view we may also require that each plant transition starts from a safe state (beyond ending up in a safe state). However a controller designed with such stronger hypothesis would stop working if for some reason the plant reaches an unsafe state. Thus for controller robustness reasons we do not assume that transitions start from safe states, we only require that transitions lead to safe states. As a result if, for some reason, the plant ends up in an unsafe state our controller will be able to trigger a transition (if any) taking the plant back to a safe state. At last we require that synchronization (`sync` in fig. 10) be satisfied. This leads us to the definition for `plant_all` in fig. 10.

7 Experimental Results

From the plant model `plant_all` and the set of final states `end_all` defined in fig. 10 we can automatically

	FSV	Ctr time	Ctr src	Ctr exe	I time	I src	blk
CSL	3 weeks for specification and verification via model checking	3 days	9 pages (CSL)	NA	1 week	7 pages (C)	4
ESTEREL	Only specification, no verification.	NA	NA	46KB	NA	NA	NA
LUSTRE	About 1 week for specification and verification using Lesar	$\approx$ 1 week	200 lines (LUSTRE) yielding 800 lines (C)	NA	$\approx$ 2 weeks	NA	5
SIGNAL	1 week for specification and partial verification using Sigali	2 weeks	1700 lines (C)	90KB	1 week	NA	NA
StateCharts	About 1 week for specification and verification via model checking	$\approx$ 2 weeks	NA	NA	NA	NA	5
TLT	NA	$\approx$ 2 weeks	NA	NA	NA	NA	8
SDL	About 1 month for specification and testing using SDT, no verification.	1 month	1800 lines (SDL92)	NA	NA	NA	NA
Tatzelwurm	About 600 lines for specification, verification done using interactive proofs.	2 weeks	250 lines (Pascal)	NA	NA	NA	6
HTTDs and HOL	About 2 months for specification and verification using interactive proofs.	NA	NA	NA	NA	NA	NA
Deductive Synthesis	More than 1 month for carrying out proofs yielding a controller.	0 days	NA	NA	NA	NA	NA
Automatic Synthesis	1 week for specification. No verification needed.	0 days	203K	197K	1 week	65K	6

Figure 12: Comparing efforts from [9] with our automatic synthesis approach.

synthesize a C program implementing a (centralized) controller $\mathcal{K}$ for our plant. This is done as shown in [12, 13]. Fig. 11 gives our experimental results. Note that columns "Plant state ..." in fig. 11 refer to the plant state space. The controller itself is a combinational circuit (implemented in C in our case) with 42 inputs (representing the plant present state) and 46 outputs (representing plant events). Column "CPU" gives the CPU time for synthesizing our controller starting from formal specs. Column "Max OBDD" is the largest OBDD created during the controller synthesis computation. Column "Ctr MUX" gives the number of multiplexers needed if we were to built our controller using a combinational circuit. Each multiplexer has two inputs, one output and one control input and implements the if-then-else corresponding to an OBDD vertex in the OBDD representation for our controller. Column "Ctr MUX" shows that our controller could easily be implemented using FPGAs. Columns "Ctr..." reports the synthesized controller C source code size, executable size and max, min, average reaction time (i.e. the time our controller takes to compute a plant command from a plant state sample given in input).

Our synthesized code is essentially a (long) list of `goto`'s (as is often the case for automatically generated code). Note however that maintainability is not an issue for our controller C code. If (formal) specifications change we just resynthesize a brand new controller. On the other hand maintainability considerations do apply to our closed loop formal specifications.

It took us about 1 man-week to write formal specifications and plant model. One more man-week was needed to write the interface to the Tcl/Tk simulator provided by FZI [9].

Fig. 12 shows effort data for the case studies in [9] for which such data where available (NA stands for "not available"). The first column identifies the case study using the name of the language or tool used to carry it out. The row labeled "Automatic Synthesis" gives data for our approach. Column "FSV" gives data on Formal Specification and Verification. Columns "Ctr time", "Ctr src", "Ctr exe" give data for, respectively, the controller development time, source code, executable code. Columns "I time", "I src" give data for, respectively, development time and source code for the interface to the FZI simulator. Column "blk" gives the maximum number of metal blanks simultaneously present in the production cell that can be correctly handled by the controller.

It must be kept in mind that the case studies in [9] were developed by teams with different levels of expertise (some where developed by students some by experienced researchers). Moreover the tools used were also at different stages of development. Thus the data in fig. 12 can only be considered as a rather rough indication. Anyway, these are the only data we have.

Our design time effort compares favourably with the time needed to carry out manual design followed by automatic verification (ranging from weeks to months in fig. 12).

Much of our formal specification work consists in building a (formal) model of the plant. Note that, no matter the design method used, simulation will always be there and to perform simulation a plant model is needed (provided by FZI in our case). In particular plant modeling has to be done even when using manual design possibly followed by formal verification.

Once a plant formal model is available synthesizing a C program implementing our embedded controller takes about 15 minutes on a Linux PC with a 200 MHz Pentium and 112MB RAM. Our controller C code size as well as executable size (fig. 11) are reasonable, although not particularly small. E.g. the controller executables obtained via manual design using Esterel or Signal have size, respectively, 46K and 90K (fig. 12).

Reaction times in fig. 11 where measured by running our controller on our Linux PC and inserting calls to the `clock()` C function in our controller code. We found no data about reactions times in [9]. Anyway all controllers in [9] as well as our are "fast enough" for the plant production cell accordingly to the FZI simulator.

Our controller works for all possible relative speeds of cell motors and can handle up to 6 metal blanks simultaneously present in the production cell. This is the same or more than many of the manually designed controllers in [9] can do. Note however that, by suitably refining requirements, the controller designed in [6] (TLT in fig. 12) can handle up to 8 metal blanks in the system.

Of course the (finite state) models we developed for feed belt, robot, table, etc can all be reused. This makes reusability high for design with automatic synthesis from formal specs (see sec. 1).

Altogether from this case study we conclude that for moderate size industrial automation plants automatic controller synthesis from formal specifications favourably competes with manual design followed by automatic verification.

To some extent this is to be expected since we are replacing the activity of writing programs with that of writing formal specifications. Since formal specifications and a plant model (to run simulation as well as model checking) have to be produced anyway, when a verified controller is required we save on the coding effort. Indeed we suspect that, with the appropriate infrastructure in place, we save on design effort even when no formal verification is needed.

Note, however, that our approach only makes sense when writing closed loop formal specifications is easier than manually designing our system and then verifying it. This is often the case for embedded control systems, but it is false in general. E.g. our approach cannot be used to design, say, a word processor.

8 Conclusions

We succeeded in automatically synthesizing a controller from closed loop formal specifications for a moderate size system (about 10^{12} states) modeling an industrial metal processing plant near Karlsruhe [9]. To the best of our knowledge this is the first case study on automatic synthesis of controllers from formal specifications for industrial automation plants.

From a verification perspective this case study has been thoroughly studied in [9]. Comparing our results (sec. 7) with those in [9] we conclude that for industrial automation control systems, automatic synthesis of embedded controllers from formal specifications is a viable and indeed profitable alternative to manual design followed by automatic verification.

We see two main obstructions to our approach: state explosion (also present in automatic verification via model checking); controller size (is larger than that obtained by using manual design, for some application this could be a problem). These obstructions are obvious subjects for our future research.

Acknowledgments

I am grateful to Victor Winter and to anonymous referees for helpful comments and suggestions about this paper.

References

[1] S. P. Khatri, A. Narayan, S. C. Krishnan, K. L. McMillan, R. K. Brayton, A. Sangiovanni-Vincentelli *Engineering Change in a Non-Deterministic FSM Setting*, Proc. of 33rd IEEE/ACM "Design Automation Conference", 1996

[2] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, L. J. Hwang, *Symbolic model checking:* 10^{20} *states and beyond*, Information and Computation 98, (1992)

[3] S. Balemi, G. J. Hoffmann, P. Gyugyi, H. Wong-Toi, G. F. Franklin, *Supervisory Control of a Rapid Thermal Multiprocessor*, IEEE Trans. on Automatic Control, Vol. 38, N. 7, July 1993

[4] R. Bryant, *Graph-Based Algorithms for Boolean Function Manipulation*, IEEE Trans. on C., Vol. C-35, N.8, Aug. 1986

[5] J. Burghardt, *Deductive Synthesis*, in [9]

[6] J. Cuellar, M. Huber *TLT*, in [9]

[7] H. Dierks, *Synthesising Controllers from Real-Time Specifications*, Proc. of 10th IEEE "International Symposium on System Synthesis", Sept. 1997, Antwerp, Belgium

[8] E. A. Emerson, *Temporal and Modal Logic*, Handbook of Theoretical Computer Science, Elsevier 1990

[9] C. Lewerentz, T. Lindner (Eds), *Formal Development of Reactive Systems: Case Study Production Cell*, LNCS 891, Springer-Verlag 1995

[10] P. J. Ramadge, W. M. Wonham, *The Control of Discrete Event Systems*, Proc. of the IEEE, Jan. 1989

[11] E. Tronci, *Hardware Verification, Boolean Logic Programming, Boolean Functional Programming*, Proc. 10th IEEE Conf. on "Logic In Computer Science" 1995, San Diego, CA, USA

[12] E. Tronci, *On Computing Optimal Controllers for Finite State Systems*, Proc. of 36th IEEE CDC, 1997, USA

[13] E. Tronci, *Automatic Synthesis of Controllers from Formal Specifications*, Proc. of 2nd IEEE Int. Conf. on "Formal Engineering Methods", Dec. 1998, Brisbane, Australia

[14] Ftp to `gate.fzi.de`, login as `anonymous`, cd to `pub/korso/fzelle/simulation` get file `visualization.tar.Z`