Journal of Universal Computer Science, vol. 3, no. 5 (1997), 603-665
submitted: 20/12/96, accepted: 20/5/97, appeared: 28/5/97

Integrating ASMs into the Software Development Life Cycle

Egon Börger
(Università di Pisa, Italy
boerger@di.unipi.it)

Luca Mearelli
(Università di Pisa, Italy
luca@tex.odd.it)

Abstract: In this paper we show how to integrate the use of Gurevich's Abstract State Machines (ASMs) into a complete software development life cycle. We present a structured software engineering method which allows the software engineer to control efficiently the *modular development* and the *maintenance* of well documented, formally inspectable and smoothly modifiable code out of rigorous ASM *models for requirement specifications*. We show that the code properties of interest (like correctness, safety, liveness and performance conditions) can be proved at high levels of abstraction by traditional and reusable mathematical arguments which—where needed—can be computer verified. We also show that the proposed method is appropriate for dealing in a rigorous but transparent manner with hardware-software co-design aspects of system development.
The approach is illustrated by developing a C^{++} program for the production cell control problem posed in [Lewerentz, Lindner 95]. The program has been validated by extensive experimentation with the FZI production cell simulator in Karlsruhe and has been submitted for inspection to the Dagstuhl seminar on "Practical Methods for Code Documentation and Inspection" (May 1997).
Key Words: Programming Techniques, Requirement Specification, Stepwise Refinement, Code Documentation, Code Inspection, Program Verification, Model Checking, Abstract State Machines.
Category: D.1, D.1.3, D.1.5, D.2.1, D.2.4, D.2.5

1 Introduction

Gurevich's Abstract State Machines (previously called evolving algebras, see [Gurevich 95]) have been used successfully to specify real-life programming languages (e.g. Prolog, C^{++}, VHDL, Oberon) and architectures (e.g. PVM, Transputer, DLX, APE100), to validate standard language implementations (e.g. of Prolog on the WAM, of Occam on the Transputer), to verify numerous distributed and real-time protocols, etc.. A survey and a methodological motivation for the new ASM approach in comparison to other specification and verification approaches can be found in [Börger 95]; see also [Börger 94], the annotated ASM bibliography [Börger 95a], the most recent work appearing in this and the previous J.UCS special ASM issue and the two ASM home pages http://www.uni-paderborn.de/cs/asm.html and http://www.eecs.umich.edu/gasm/.

In this paper we investigate the idea to use stepwise refinement of ASMs for modular development of well documented, formally inspectable and feasibly modifiable code out of ASM models for requirement specifications. We illustrate the proposed structured software engineering approach by developing the automation software for a reactive distributed system, namely a C^{++} program to control the production cell introduced in [Lewerentz, Lindner 95] as case study derived from "an actual industrial installation in a metal-processing plant in Karlsruhe" to obtain a "realistic, comparative survey" for testing "the usefulness of formal methods for critical software systems and to prove their applicability to real-world examples" [Lindner 95]. The specification together with its refinement to executable code (see [Mearelli 97]) have been submitted for inspection to the participants of the Dagstuhl seminar on "Practical Methods for Code Documentation and Inspection" organized by D. Parnas, P. Joannou and the first author from May 12-16, 1997.

1.1 The Main Development Steps

Specification. As starting point we define a *ground model* (in the sense of [Börger 94]) which is rigorous but transparent and concise and can be convincingly shown to faithfully reflect the production cell as informally specified in [Lindner 95]. We explain how the information hiding and abstraction mechanisms of ASMs allow the designer to formulate such ground models as an appropriate interface to the real world as it is perceived by the customer—a feature which presupposes in particular that the ground model is expressed in terms of the given application domain. The standard notation used for ASMs makes it possible to come up with transparent, application driven descriptions which provide a realistic chance to expose formalization errors. Moreover information hiding and abstraction mechanisms of ASMs encourage and facilitate the decomposition of the system into components and their modular development, based on a transparent definition of *precise interfaces* through which the components are put together (at various levels of abstraction). The definition of these interfaces is obtained by a faithful formalization of the distributive features of the problem as given by the informal task description and yields a clean separation of the local actions of each simple component from the cooperation activities between different components. It is interesting that this interface definition together with the abstract ground model definition of the functional behaviour of each single component suffices for a proof of the strong liveness property required in the informal task description, see the Agent Progress Lemma below.

Design. Starting from this abstract model we define a *series of refined models* leading to C^{++} code which has been validated by extensive experimentation showing that the program controls successfully the production cell simulation environment at FZI Karlsruhe. The ability of ASMs to be easily tailored to any abstraction level facilitates the definition of refinement steps which avoid premature design decisions and make each design decision transparent. This fits well the common experience that a good refinement hierarchy comes out only after various attempts to structure the design in an appropriate way.

Analysis. The simple mathematical relations between the various ASM models allow the designer to produce rigorous arguments for the correctness of his design, namely to *prove the system properties of interest* (safety, liveness, maximal performance etc.) by first establishing them under natural assumptions at high levels of abstraction and then showing that each refinement step preserves those assumptions. In other words one can control the code development at high levels of abstraction by proving the required code properties for abstract code through drawing simple conclusions from assumptions which are related in a natural way to basic conditions of the underlying problem (domain)—and therefore not yet complicated by later design decisions—and which can easily be shown to be preserved by the subsequent design steps. The abstraction possibilities offered by ASMs yield a reduction of the proofs to simple proofs of local conditions on single machines or on groups of communicating machines or of machines which are related by a refinement step. It is important that the simplicity of the ASM models allows one to establish these proofs in ordinary mathematical terms, as they are familiar to engineers and programmers, so that during the program development the working computer scientist can check and guarantee by himself the properties of interest. The value of this feature offered by the ASM approach is enhanced by the fact that those traditional mathematical proofs come in a form which makes them reusable when the abstract models and the code evolve due to changing requirements. This is particularly helpful for keeping control of the functionality of the code during the evolutionary maintenance process.

The possibility to integrate standard system engineering reasoning into the ASM modeling activity as a means to keep control of the overall system development is one of the reasons for the practicality of the method. It has to be judged against the well known fact that purely mechanical methods like logical deduction (theorem proving) and symbolic model checking face scalability problems for applications which are beyond the complexity of relatively simple cases like the production cell (see the evidence reported in [Lewerentz, Lindner 95b]). However standard mathematical proofs accompanying the development of stepwise refined ASMs are amenable to further detailing through mechanization in machine based proof systems. An illustrative example for such an endeavour is the complete verification of the ASM based WAM correctness proof in [Börger, Rosenzweig 94] which used the interactive theorem prover KIV and is partially reported in this volume (see [Schellhorn, Ahrendt 97]); for another machine verification of this proof which uses ISABELLE see [Pusch 96]. Kirsten Winter has investigated for the first time the possibility to turn ASMs into finite automata in order to machine check the correctness of the ASM specification by applying advanced model checking techniques (see [Winter 97] in this volume). The same idea to use abstraction to contain the state explosion for model checking has been applied recently to the SCR method (see [Bharadwaj,Heitmeyer 97]).

The hierarchy of refined models, together with the proofs of the correctness of the implementations relating the different levels, constitute a full documentation of the result of the whole structured software development and make the executable code amenable to rigorous inspection. This combination of modular development with controlled stepwise refinement (including optimizations) provides also an economical way to achieve extendability and modifiability for the design of complex systems where cost effective maintenance and evolution of the code is an issue.

1.2 The Lesson for Software Engineering

We hope to illustrate, through a full treatment of the production cell leading from capturing requirements to the design of code, that the ASM method can be integrated with advantage into a complete software development life cycle. The method makes it possible to solve the ground model problem (providing means for capturing informal requirements by appropriate—correct, concise, transparent and flexible—formalizations); it supports incremental development by the systematic use of stepwise refinement; it allows one to simulate abstract models to validate these specifications (see [Päppinghaus 97] where the Paderborn machine [Del Castillo et al. 96] for executing ASMs is used); it shows how to turn the informal reasoning and the application domain driven explanations—which necessarily accompany every design—into a precise mathematical form which makes them accessible to (mental or machine) falsifiability experiments; it can integrate the use of machine support (type/model checking, theorem proving) for detecting errors in the formal system analysis (specifications and proofs). The ASM method for structured software engineering therefore respects the guidelines for applying formal methods proposed in [Heitmeyer 97]. Since ASMs use only the standard language and standard methods of programming and mathematics so that their use can be learnt in a couple of days by every experienced programmer, their integration into the software development life cycle can be realized for the *normal* development work avoiding the awkward idea to set up separate formal specification teams.

This point can be further clarified by comparing it to Anthony Hall's recent statement in his invited lecture at the 1997 Z User Meeting where he said that "the most important characteristic of Z, which singles it out from every other formal method, is that it is completely divorced from computation" which "means that you can use logic (otherwise known as ordinary language) to define your requirements" (see [Hall 97]). Paraphrasing this we would say that the most important characteristic of the ASM approach, which really singles it out from every other formal method, is that it allows us to happily marry the use of logic and of *abstract* computations. This marriage helps to overcome the devastating belief—which theoreticians have greatly succeeded to impose on the (above all theoretical) computer science world for decades now—into an alleged dichotomy between *declarative* (or logical) and *operational* methods. The "argument" which is usually put forward to make us regard declarative methods as good and operational ones as bad is that logic allows us to express elegantly and succintly "what" we want without having to worry about the "how" to achieve this goal whereas operational descriptions unavoidably drive us away to think about control and implementation details which are irrelevant to the goal of the high level description. This is not the place to discuss the reasons why this argument is misleading when we are looking for an appropriate treatment of the dynamics of complex computation systems (see [Börger 95]); here we have to limit ourselves to remarking that the belief in this dichotomy has proved not to be helpful for filling, in any rigorous but nevertheless practical way, the huge gap between abstract system views and their implementations.

ASMs allow the system designer to fill this gap in a controllable—theoretically well founded but nevertheless practical—way. The happy marriage ASMs produce between logic and (abstract) computation provides the software engineer with a method to define unambiguous but comprehensible and concise requirement and system specifications and to relate them (by stepwise refinement of ASMs) in a direct and feasably controllable way to more detailed design levels and eventually down to executable code. This possibility to directly relate rigorous models at the levels of abstraction of the software development cycle has a considerable advantage in particular because nowadays most real systems are implemented using concurrency. Supported by distributed ASMs (i.e. ASMs with an underlying notion of concurrent computation) we can avoid doubling the specification work when it comes to relate the specification in a reliable way to the design; again the comparison to Z taken from Anthony Hall's invited lecture at ZUM'97 is illuminating: "Going...towards design we have to recognize that Z is simply not the right language to use...we need to go from our Z specification, through some computational model of the specificand (such as an action system) to a refinement in, for example, a concurrent version of the refinement calculus. Of course we may find ourselves re-introducing Z or Z-like specifications of modules in the design..."(see [Hall 97]).

At first sight the length of this paper seems to confirm the unavoidability of combinatorial explosion which most formal methods experience when applied to more than academic examples, although in publications the phenomenon often remains hidden behind the argument that "due to the lack of space and time the details have to be skipped". We want to demonstrate to the reader who is willing to have a closer look that the proposed use of ASMs for structured software development *is* practical and in particular can avoid the combinatorial explosion even if all the necessary details are spelled out in full. Therefore this paper develops, from scratch, the entire *formal specification*, the complete *code design* and the *analysis* (with detailed proofs for all the required system properties), including citations of those parts of the informal task description which are relevant for convincing the customer that the ground model problem has been solved (besides pictures visualizing all the rules, rule summaries and extensive comments explaining the salient features of the ASM method and comparing it to competing approaches). Alltogether this is not more than what one has to expect, for a program with 1000 lines of code, from a professional documentation which a) explains to the customer that and how the original task is accomplished by the code, b) contains a software reference manual documenting all the design decisions and the interface conditions which are needed for the maintenance of the program and in particular for cost effective and reliable program evolution.

The reader who is only interested in the specification method should skip the proofs of the required system properties and the section on the code and may jump to the program summaries in the appendices once he has understood the method from the first component machines.

2 Notation and Prerequisites

This section can be skipped to be consulted should the necessity arise. It will help if the reader is familiar with the semantics of *Abstract State Machines*, defined in [Gurevich 95]. We could have used also Parnas' tabular notation [Parnas, Madey 95] for which a simple but rigorous semantics can be given in terms of ASMs (see [Börger 96]). However what follows can be understood correctly also by reading our ASM rules as pseudo–code over abstract data types. We therefore point here only to some of the basic ASM features.

A distributed ASM is given by a set of agents and a program function *Mod* which assigns to each agent a module ("sequential" program) consisting of a finite number of so called transition rules of the following form:

$$\text{If } Cond \text{ then } Updates$$

where *Cond* is any expression (of first order logic) and *Updates* is a finite set of function updates, i.e. of updates $f(t_1, \ldots, t_n) := t$.

The *states* of ASMs are arbitrary structures, i.e. domains with predicates and functions defined on them (where without loss of generality we treat predicates as characteristic functions). The collection of the types of the functions (and predicates) which can occur in a given ASM is called its *signature*. The computational meaning of an ASM M is that given any state S (of the signature of M), for each transition rule such that *Cond* is true in S, all the updates $f(t_1, \ldots, t_n) := t$ in the set *Updates* of that rule are executed simultaneously, i.e. the value of function f at the given argument combination $t_1, \ldots, t_n$, computed in S, is changed to the value t which has been computed in S. The result of this computation step is a new state which differs from S only by some values for some of the functions where the 0-ary functions play the role of the usual programming variables. (The simultaneous execution of all the updates in a rule abstracts from intermediate copying to save some values which are updated but are also needed to compute the values for some other update.)

Each agent of a distributed ASM fires its rules at its own "time"; the overall distributed computation (a "run") is a graph made up from computation sequences of the single sequential agents which may be synchronized through shared functions (the *interaction* functions, see below). Often it is possible without loss of generality to adopt the eagerness assumption of synchronized languages, i.e. that whenever a rule is enabled it is immediately applied. (For the production cell ASM specification below we can make this assumption.) ASMs usually come together with a set of *integrity constraints* (on the domains, functions, rules) and with initialization conditions representing assumptions on the intended computations. This intuitive idea of runs of distributed ASMs should suffice for the purposes of this paper. For a precise definition see [Gurevich 95].

For a good understanding of how the distributed nature of the production cell is reflected in our models we define here the following classification of functions which is suggested by the concept of ASMs and has proved to be particularly convenient for applications. Let an ASM M be given. What follows is to be understood with respect to M. Functions can be either *static*—i.e. never changing during any run of M—or *dynamic* (otherwise). Dynamic functions may change during a run of M "as a consequence of" updates by M or updates by the environment (i.e. by some other agent than M). This results in the distinction of the following four subclasses of dynamic functions, called controlled, monitored, interaction and derived functions respectively. *Controlled* functions (for M) are dynamic functions which are directly updatable by and only by the rules of M, i.e. functions f which appear in a rule of M as leftmost function (namely in an update $f(s) := t$ for some s, t) and are not updatable by the environment. *Monitored* functions are dynamic functions which are directly updatable by and only by the environment, i.e. which are updatable but do not appear as leftmost function in updates of M. Monitored and controlled functions are generalizations of the controlled and monitored variables in the Parnas-Madey Four Variable Model (see [Parnas, Madey 95]).

Interaction functions are dynamic functions which are directly updatable by rules of M and by the environment. They are particularly useful for decomposing specifications of complex systems into simpler components because they allow the designer to separate the dynamic aspects he wants to focus on from other dynamic aspects from which he can abstract by relegating them to environmental updates. In doing so one has to keep in mind to formulate precise protocols regulating the mixed access rights to guarantee the consistency of updates of such functions by different agents (M and the environment). *Derived* functions are dynamic functions which are not directly updatable neither by M nor by the environment but are nevertheless dynamic because defined (for example by an explicit definition or by an inductive definition) in terms of static *and* dynamic functions. Derived functions can be classified into indirectly controlled, indirectly monitored or indirect interaction functions depending on the nature of the auxiliary functions used for their definition.

Updatable functions are controlled or interaction functions, *non updatable* functions are static, monitored or derived. In applications it is sometimes useful to adopt the preceding classification with respect to update locations (i.e. particular arguments of a function) or with respect to specific function values (see [Börger, Mazzanti 97] for an example), but we will not need this for the production cell.

For boolean-valued functions b we often write *b(x)* instead of *b(x)=true* and *not b(x)* or $\neg b(x)$ instead of *b(x)=false*. Sometimes we will take advantage of the *Self* function introduced in the definition of distributed ASMs in [Gurevich 95] to parameterize the agent specific functions. When the agent *Self* is clear from the context, we will omit mentioning it. An example is the function pair Bottom/TopPosition which appears below with parameters ERT (for the elevating rotary table in the production cell) and Press (for the press).

For the pictorial representation of our rule systems we use the standard notation from flowcharts and finite automata where states are visualized by circles, firing conditions by rhombs and updates by rectangles. One can adopt automatic translations between these semantically equivalent notations.

3 The Ground Model

The purpose of the first production cell model GroundCELL, a ground model in the sense of [Börger 94], is to produce an application oriented rigorous formulation of the informal description which allows one to *justify* that this formalization provides a correct model of the desired system. The most general abstraction mechanism and the flexibility of the ASM language make it easy to express such ground models in terms of the given application domain so that one is enabled to discuss with the customer, by comparison of the ASM ground model with the informal description, that his requirements are met. The ground model can then be used as the basis for the subsequent stepwise refinements and extensions leading to executable code.

We start in an object-oriented spirit by defining what are the basic objects composing the system, their basic operations and interactions. Our method is to extract these items from the informal task description (see [Lindner 95]).

> *. . . the production cell is composed of two conveyor belts, a positioning table, a two-armed robot, a press, and a travelling crane. Metal plates inserted in the cell via the feed belt are moved to the press. There, they are forged and then brought out of the cell via the other belt and the crane.[Lindner 95]*

Accordingly we specify the system as a distributed ASM with six modules, one for each of the *agents* composing the production cell and working together concurrently where each of the component ASMs follows its own clock. We will see below that each of the agents represents a sequential process which can—and under the Cell Assumption does without loss of generality—execute its rules as soon as they become enabled. The sequential control of each agent is formalized using a function currPhase : Agent→Phase which yields at each moment the *current phase of the agent* (characterizing the action this agent is going to perform). We write currPhase for the 0-ary controlled function currPhase(Self) when the agent *Self* is clear from the context.

The dependencies, among the various parts of the cell and between the cell and the environment, are formalized in the ground cell in such a way that each of these parts is defined with a precise interface to the rest of the system—typically through functions which are monitored for the single parts and represent for these parts the embedding environment. Through the ground model we define the interfaces for each agent and describe the basic actions of each module machine without detailing the way this action is performed. In order to be able to prove, at this level of abstraction, the required safety conditions (locally for each component machine) and the liveness of the system (globally as a property of the interactions among the machines composing the cell), we explicitly formulate appropriate assumptions from which these properties follow easily. In order to preserve these properties through the later refinements it suffices then to verify these assumptions at the refinement level.

We also have to make plausible assumptions on the physical characteristics of the production cell devices and of their actions because the informal description says nothing about their speed, dimension, etc. We abstract from the (finite) duration of actions performed by the devices. This is in accordance with what happens in the simulator (where the actuators are just started and stopped by the controller to trigger or halt the devices) and in accordance with the fact that ASM rules are atomic, i.e. "executed in zero time" (although there are natural ways to describe durative actions with ASMs, see [Börger et al. 95] where durative actions of distributed agents are reduced to atomic actions). Similar assumptions are that the (finite) belts have at least the space to hold two pieces each, that every object which keeps moving will eventually arrive at its destination position and will be detected there by the corresponding sensor ("device liveness"), that sensor signals arise only when an object is moving into the correponding position and stay as long as the object does not leave that position, etc. We refer to such assumptions generically as *Cell Assumption*. For the geometrical layout of the production cell see the picture in the appendix.

All the approaches reported in [Lewerentz, Lindner 95] to which we suggest to compare our solution assume that the reaction of the control software is sufficiently fast to fulfill the appropriate timing requirements. Therefore we too make this assumption. This simplifies the task although ASMs have no difficulty of principle to deal with real-time conditions (see [Gurevich, Huggins 96] where ASM agents perform instantaneous actions in continuous time).

In the following subsections we define separately each of the six sequential component ASMs which, put together as distributed ASM, constitute the ground cell GroundCELL. The initialization conditions which we impose on the ground model are motivated by the desire to avoid a certain number of tedious case distinctions in the safety and liveness proofs. We will see below that these assumptions can be guaranteed by a standard preprocessing technique.

3.1 The Feed Belt Ground Model

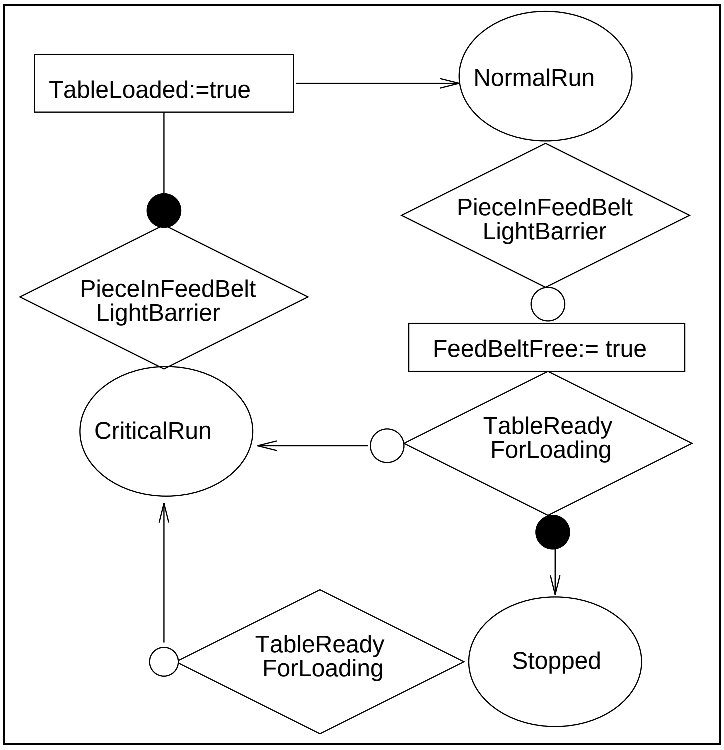
> *...The task of the feed belt consists in transporting metal blanks to the elevating rotary table. The belt is powered by an electric motor, which can be started up or stopped by the control program. A photoelectric cell is installed at the end of the belt; it indicates whether a blank has entered or left the final part of the belt. ... the photoelectric cells switch on when a plate intercepts the light ray. Just after the plate has completely passed through it, the light barrier switches off. At this precise moment, the plate ... has just left the belt to land on the elevating rotary table—provided of course that the latter machine is correctly positioned ... the feed belt may only convey a blank through its light barrier, if the table is in loading position ... do not put blanks on the table, if it is already loaded ... [Lindner 95]*

We formalize this description by abstracting from the motors (see the remark in the section on the press ground model) which yields an automaton with three states (phases). In phase NormalRun the automaton is *"transporting metal blanks to the elevating rotary table"* none of which has yet *"entered the final part of the belt"*. This phase can and will change to CriticalRun or Stopped when the photoelectric cell, formalized by a 0-ary boolean-valued monitored function PieceInFeedBeltLightBarrier , switches to the value true and thereby indicates that *"a blank has entered ... the final part of the belt"*. The automaton switches to phase CriticalRun if the elevating rotary table is *"ready for loading"*, i.e. if the value of the 0-ary boolean-valued derived function TableReadyForLoading is true which is defined by the table being in load position and not loaded. If the elevating rotary table is not *"ready for loading"*, the feed belt is Stopped and can continue later in CriticalRun only after the elevating rotary table has become ready for loading. The feed belt leaves its CriticalRun phase—and indeed goes back to NormalRun —when the photoelectric cell, by switching the value of PieceInFeedBeltLightBarrier from true to false, indicates that *"a blank has ... left the final part of the belt"*, i.e. that the table has been loaded with that blank. We have chosen to let the feed belt run also when it carries no piece, in accordance with one of the options of the informal specification.

The feed belt "communicates" that the table is loaded by setting the 0-ary boolean-valued function TableLoaded (i.e. by updating it from false to true) when dropping a piece. This function is monitored for the elevating rotary table and an interaction function for the feed belt (and for the robot which will only reset the function—update it from true to false—, namely when retrieving a piece from the table. This guarantees the consistency of this function.)

In the informal specification there is no sensor at the beginning of the belt to determine whether the feed belt is free to receive a piece. It is said that *" a new blank may only be put on the feed belt, if ... the last one has arrived at the end of the feed belt"*. We interpret this as allowing (at most) two pieces on the belt at any time and formalize the control of the deposit of blanks on the feed belt by a 0-ary interaction function FeedBeltFree which is updated by the feed belt from false to true when *"a blank ... has arrived at the end of the feed belt"*. We will see below that the only other agent which can update this function is the traveling crane, namely by switching it from true to false when a blank is dropped onto the feed belt. This will guarantee the consistency of the updates of this function.

The preceding formalization is summarized by the following ASM. ERT stands for the elevating rotary table.



[Feed Belt]

FB_NORMAL.
if currPhase = NormalRun and PieceInFeedBeltLightBarrier
then FeedBeltFree := True
    if TableReadyForLoading then currPhase := CriticalRun
                                else currPhase := Stopped

FB_STOPPED.
if currPhase = Stopped and TableReadyForLoading
then currPhase := CriticalRun

FB_CRITICAL.
if currPhase = CriticalRun and not PieceInFeedBeltLightBarrier
then currPhase := NormalRun
    TableLoaded := True

where TableReadyForLoading ≡ TableInLoadPosition and not TableLoaded
    TableInLoadPosition ≡ currPhase(ERT) = StoppedInLoadPosition

For the initialization we assume currPhase = NormalRun , FeedBeltFree = true and consequently PieceInFeedBeltLightBarrier = false.

The lack of a photoelectric cell or of a traffic light at the beginning of the feed belt makes the feed belt ASM asymmetric to the deposit belt ASM below; a simpler uniform feed and deposit belt machine would result from the decision to have such blank detectors at the beginning and at the end of the belts. We leave it as an exercise to play with the simple variations which suffice to reflect such hardware/software co-design decisions and to compare their result.

We illustrate that the feed belt safety property required by [Lindner 95] can be easily proved at the level of abstraction of this ground model ASM Feed Belt.

Feed Belt Safety Property . *The feed belt does not put metal blanks on the table if the latter is already loaded or not stopped in loading position.*

Proof. A piece can be dropped onto the elevating rotary table only by executing the feed belt rule FB_CRITICAL. This rule is applicable only if the feed belt is in phase CriticalRun which can be entered only by applying FB_NORMAL or FB_STOPPED when TableReadyForLoading is true, i.e. by definition when the table is not yet loaded and is stopped in loading position.

3.2 The Elevating Rotary Table Ground Model

> *... The task of the elevating rotary table is to rotate the blanks by about 45 degrees and to lift them to a level where they can be picked up by the first robot arm. The vertical movement is necessary because the robot arm is located at a different level than the feed belt and because it cannot perform vertical translations. The rotation of the table is also required, because the arm's gripper is not rotary and is therefore unable to place the metal plates into the press in a straight position by itself. [Lindner 95]*

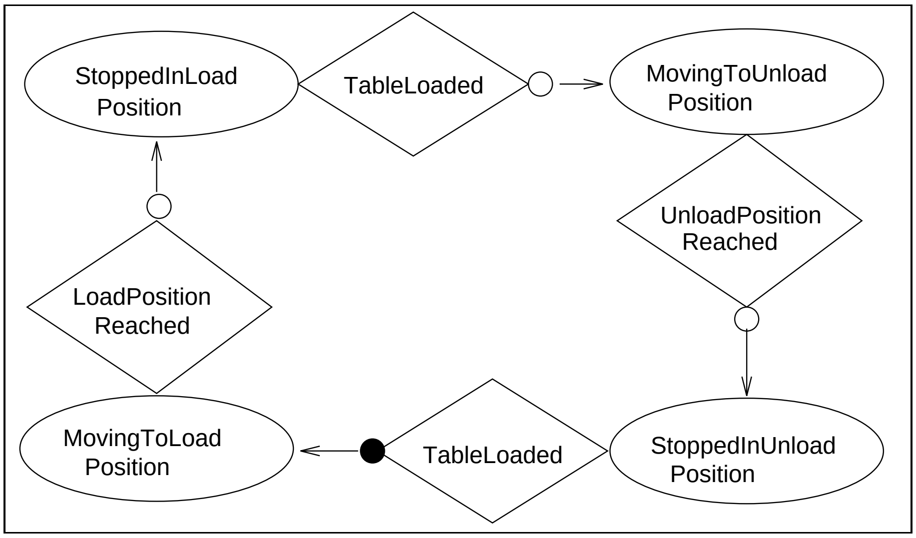
In the ground model for the elevating rotary table we abstract from the motors and from the particular movements (rotation and lifting) which are performed in order to bring each blank from the position where it has been loaded by the feed belt to the position where it can be unloaded by the first robot arm (see the remark on this abstraction in the section on the press ground model). We formalize this by two 0-ary monitored functions LoadPositionReached and UnloadPositionReached (which by the cell assumption are true iff the table is in the respective position).

As a consequence the elevating rotary table becomes a simple automaton which cycles between being StoppedInLoadPosition (waiting for being loaded by the feed belt) and being StoppedInUnloadPosition (waiting for being unloaded by the robot) by going through the intermediate phases MovingToUnloadPosition and MovingToLoadPosition . The phase changes are determined by the values of the monitored functions TableLoaded and (Un)LoadPositionReached .

The preceding formalization is summarized by the following ASM.

The table is required (to be shown) not to move over the limits represented by the load and unload positions in order not to collide with the neighbours. Although the level of abstraction choosen here is good to show the interaction with the rest of the cell, it is not detailed enough to express and prove this safety property by more than just saying that the moving phases terminate as soon as the goal positions are reached. We therefore postpone this issue to the refinement step where we will speak explicitly about the two table movements.

For the initialization we assume currPhase = StoppedInLoadPosition and TableLoaded = false. It is easy to check by inspection of the rules that after each cycle the elevating rotary table returns to this situation if a new piece arrives at the feed belt end.



[Elevating Rotary Table]

WAITING_LOAD.
if currPhase = StoppedInLoadPosition and TableLoaded
then currPhase := MovingToUnloadPosition

MOVING_UNLOAD.
if currPhase = MovingToUnloadPosition and UnloadPositionReached
then currPhase := StoppedInUnloadPosition

WAITING_UNLOAD.
if currPhase = StoppedInUnloadPosition and not TableLoaded
then currPhase := MovingToLoadPosition

MOVING_LOAD.
if currPhase = MovingToLoadPosition and LoadPositionReached
then currPhase := StoppedInLoadPosition

Remark. The reader may have noticed that from the point of view of the mere transportation of blanks, we could have pushed further the abstraction from the table movements by avoiding any mentioning of moving at all. This would yield an elevating rotary table with only two states StoppedIn(Un)LoadPosition and only two rules (resulting from the above four rules by eliminating the moving rules and by updating the currPhase directly from StoppedInLoadPosition to

StoppedInUnloadPosition and vice versa). Going from the 2-states 2-rules ASM to our 4-states 4-rules ASM would become a simple refinement step (where the WAITING-(UN)LOAD rule is mapped to the corresponding rule sequence WAITING-(UN)LOAD, MOVING-(UN)LOAD). A similar remark applies to the ground model for the robot, the press and the traveling crane.

3.3 The Robot Ground Model

We first specify the Robot ASM and then prove for it the safety properties required by the task description.

3.3.1 The Specification

> *...The robot comprises two orthogonal arms. For technical reasons, the arms are set at two different levels. Each arm can retract or extend horizontally. Both arms rotate jointly. Mobility on the horizontal plane is necessary, since elevating rotary table, press, and deposit belt are all placed at different distances from the robot's turning center. The end of each robot arm is fitted with an electromagnet that allows the arm to pick up metal plates. The robot's task consists in: taking metal blanks from the elevating rotary table to the press; transporting forged plates from the press to the deposit belt. [Lindner 95]*

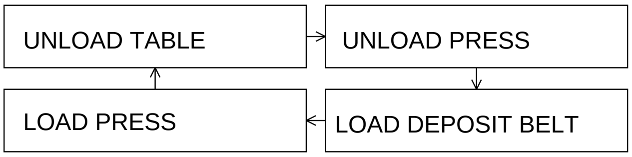
Abstraction from the motors and from the details of the movements yields four basic robot actions: unload the table/press and load the press/deposit belt. For each action the robot has to wait, in the position where the action will take place, until the conditions for the action to start are verified, then it will perform the action, and finally it has to move to the position of the next action. This can be formalized by a small automaton of four groups of three rules each controlling the passage of the robot from one action to the next.

Since we do not know the duration neither of the *actions* nor of the movements, we formalize their termination condition by corresponding boolean-valued 0-ary monitored functions *action*Completed and *action*PosReached.

In the given task description the table, the press and the deposit belt have no sensors to indicate whether they are loaded or not. This is reflected in our formalization by the fact that a) TableLoaded , PressLoaded are seen by the table and the press as a monitored function (which is controlled or an interaction function for other agents of the production cell), and b) DepositBeltReadyForLoading is an interaction function for the robot and the deposit belt. The robot communicates to the table and to the press when they have been unloaded by resetting TableLoaded and PressLoaded respectively, similarly it communicates to the press and to the deposit belt when they have been loaded by setting PressLoaded and by resetting DepositBeltReadyForLoading . (DepositBeltReadyForLoading is updated to false by the robot on exit from the phase LoadingDepBelt into which it enters only if DepositBeltReadyForLoading = true. We will see below that the only other device which can update DepositBeltReadyForLoading is the deposit belt which can update it only to true when it is false. This guarantees the consistency of the updates for DepositBeltReadyForLoading .)

Remark. How to provide the information on the load status of table, press and belts is really a hardware/software co-design question. Other decisions than the one taken by the given task description could be easily formalized and compared by slight changes of our ASM model. For example adding a sensor to the table and cancelling the updates of TableLoaded yields a model where TableLoaded is monitored by the robot, the feed belt and the table.

We have to decide upon the order of the robot actions: the choice depends on many issues such as efficiency considerations (maximal work of the press or of the robot, maximal throughput, and so on) or the geometrical placement of the machines that interact with the robot, or the simplicity of the controller. In the task description [Lindner 95] a solution is suggested that ensures minimal movements of the robot; the sequence of movements is determined on the basis of the relative positions of table, robot, press and deposit belt, and apparently yields a good usage of the press, letting the robot be ready to load the press whenever it is unloaded. We adopt the suggested order for our model: *1. Unload Table: ... picks up a metal blank from the elevating rotary table, 2. Unload Press: ... picks up a forged work piece, 3. Load Deposit Belt: ... places the forged metal plate on the deposit belt* (as soon as there's enough space at the belt end), *4. Load Press: ... deposits the blank in the press.*



The preceding formalization is summarized by the ASM below. We anticipate the definition of Table(Press)In(Un)LoadPosition. For the initialization we assume that the press is initially loaded with a piece and that the robot is in the WaitingInUnloadTablePos phase. | separates different cases.

[Robot]

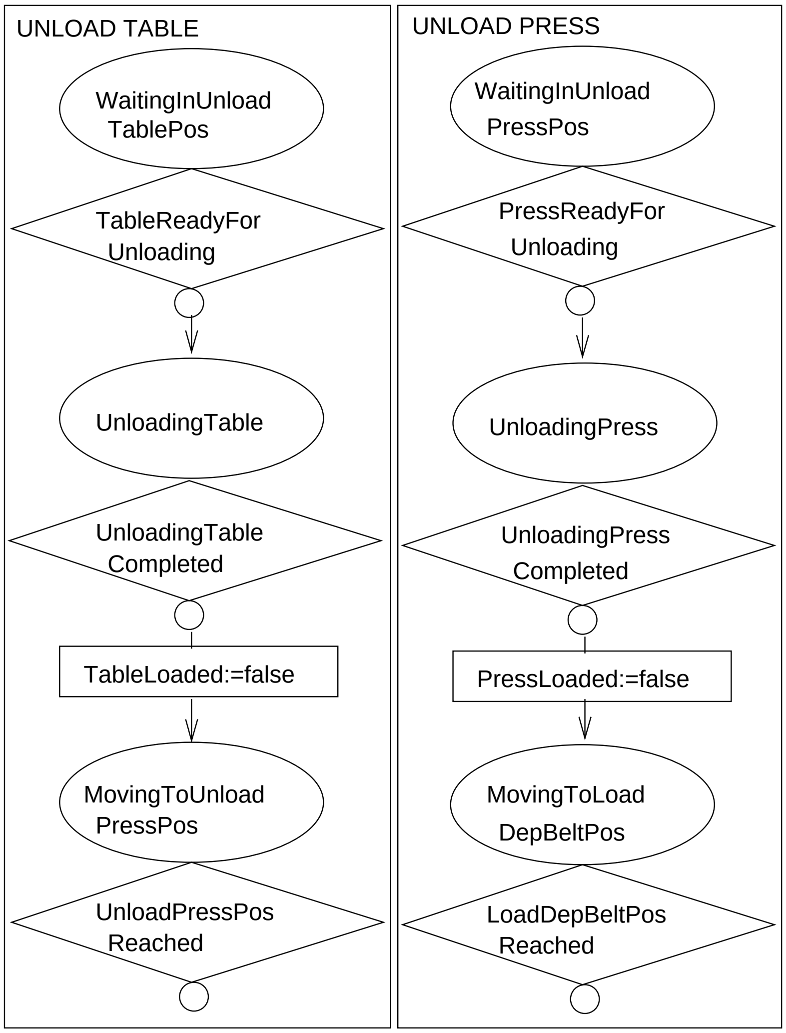
WAITING.
if currPhase = WaitingIn(UnloadTable|UnloadPress|LoadDepBelt|LoadPress)Pos
 ∧ (Table|Press|DepositBelt|Press)ReadyFor(Unloading|Unloading|Loading|Loading)
then currPhase := UnloadingTable |UnloadingPress |LoadingDepBelt |LoadingPress

ACTION.
if currPhase = UnloadingTable |UnloadingPress |LoadingDepBelt |LoadingPress
 ∧ (UnloadingTable |UnloadingPress |LoadingDepBelt |LoadingPress)Completed
then currPhase:=MovingTo(UnloadPress|LoadDepBelt|LoadPress|UnloadTable)Pos
    TableLoaded |PressLoaded |DepositBeltReadyForLoading |PressLoaded :=
      false|false|false|true

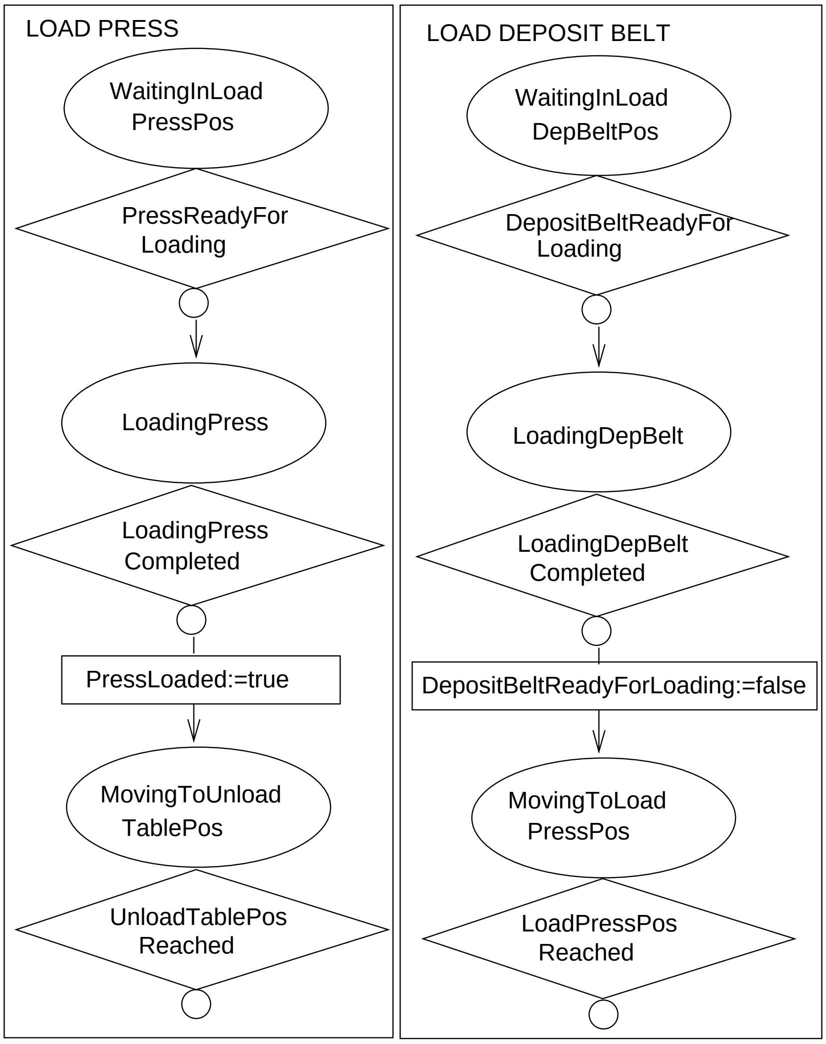
MOVING.
if currPhase = MovingTo(UnloadPress|LoadDepBelt|LoadPress|UnloadTable)Pos
  and (UnloadPress|LoadDepBelt|LoadPress|UnloadTable)PosReached
then currPhase:=WaitingIn(UnloadPress|LoadDepBelt|LoadPress|UnloadTable)Pos

where TableReadyForUnloading ≡ (TableInUnloadPosition and TableLoaded)
    TableInUnloadPosition ≡ (currPhase(ERT) = StoppedInUnloadPosition)

PressReadyForUnloading ≡ (PressInUnloadPosition and PressLoaded)
PressInUnloadPosition ≡ (currPhase(Press) = OpenForUnloading)
PressReadyForLoading ≡ (PressInLoadPosition and not PressLoaded)
PressInLoadPosition ≡ (currPhase(Press) = OpenForLoading)



Remark. The above choice for the order of the robot actions is responsible for the Last Piece Problem discovered in [Nökel, Winkelmann 95] when trying to model check that when the press holds a piece, the robot will eventually unload it from the press. In our formalization this problem becomes apparent in the ground model as resulting from the decision about the order for robot actions, namely that if there is a last blank which is loaded to the press, then the robot will not be able to unload it from the press because after having loaded the press with the last blank the robot will go to unload the table and stay waiting forever. One way to avoid the problem is to allow a nondeterministic choice for the order of the robot actions (with the side effect that the maximal throughput of blanks is 8 instead of 7, see the TLT solution in [Lewerentz, Lindner 95] and the proof for the strong performance property below).



3.3.2 The Safety Properties

At this level of abstraction we can state robot safety requirements by defining abstract functions modeling the physical movements of the robot (rotating its base, extending/retracting its arms and switching on/off its magnets) and prove them from assumptions which relate the values of these functions to the robot phases. These assumptions will then be used as integrity constraints to guide the refinement steps. The reader who is interested only in the specification and not in the proofs of the system properties may skip this section.

The description given in [Lindner 95] for the rotation operations performed by the robot arms can be axiomatized by using two functions RobotRotationMot (describing the robot rotation, i.e. the action of the motor driving the robot base) and Angle (for the potentiometer indicating how far the robot has rotated). From the geometric placement of the components described by the informal specification we see that the bounds to the rotation of the robot are represented by the position where the first robot arm points towards the elevating rotary table (LeftRobotBound, i.e. the value reached when Angle = Arm1ToTable) and by the position where the first robot arm points towards the press (RightRobotBound, i.e. the value reached when Angle = Arm1ToPress). The movements taking place between these two bounds are reflected by axioms describing how the value of Angle increases or decreases between these bounds. We summarize these conditions in the following two assumptions.

Robot Assumption 1 .

currPhase = MovingTo(UnloadPress|LoadDepBelt|LoadPress)Pos
    ⇒ RobotRotationMot = counterClock
currPhase = MovingToUnloadTablePos ⇒ RobotRotationMot = clockwise

currPhase = WaitingIn(UnloadTable|UnloadPress|LoadDepBelt|LoadPress)Pos
∨ currPhase = UnloadingTable | UnloadingPress |
        LoadingDepBelt | LoadingPress
    ⇒ RobotRotationMot = idle

Angle increases if RobotRotationMot = counterClock, decreases if RobotRotationMot = clockwise, doesn't change if RobotRotationMot = idle

Robot Placement Assumption .

LeftRobotBound = UnloadTablePos
    ≤ UnloadPressPos ≤ LoadDepBeltPos
    ≤ LoadPressPos = RightRobotBound
UnloadTablePosReached ⇔ Angle = Arm1ToTable
UnloadPressPosReached ⇔ Angle = Arm2ToPress
LoadDepBeltPosReached ⇔ Angle = Arm2ToDepBelt
LoadPressPosReached ⇔ Angle = Arm1ToPress

From the initialization condition Angle = Arm1ToTable and these two assumptions we can prove the robot safety requirements that the robot must not be rotated clockwise, if arm 1 points towards the elevating rotary table, and it must not be rotated counterclockwise if arm 1 points to the press [Lindner 95].

Robot Safety Property 1 . *The robot never rotates over its bounds.*

Proof. By induction on robot runs. For the base of the induction the claim holds by initialization. By Robot Assumption 1 , the value of Angle can change only if RobotRotationMot = idle is not true, i.e. during the MovingTo[...] phases. Those phases are terminated by the MOVING rules as soon as the robot reaches the intended positions (which by Robot Assumption 1 and by the Robot Placement Assumption respect the robot bounds).

In order to state conditions guaranteeing the avoidance of collisions between robot arms and their neighbour components define Arm*i*Ext as the sensor function (potentiometer) describing the extension of the i-th robot arm, to be measured as a number indicating the distance of the magnet from the rotating center of the robot. Consider the safety request that the robot should not crash with the press. Such a collision would happen would the robot extend too much its arms while pointing them towards the press or had the robot an arm inside the press when the press is closing. The collisions can be avoided if one allows the robot to move (i.e. actually to rotate) near the press only if its arms are retracted enough. Similar conditions are required for the avoidance of collisions between the first arm and the table, and between the second arm and the deposit belt:

> *...In order to meet the various safety requirements ... a robot arm must retract whenever a processing step were it is involved is completed. [Lindner 95]*

Stated otherwise, the arms should extend only when required to pick up or to drop a piece. We abstract from the geometrical details of a definition of "in the proximity of the press, table, deposit belt" by imposing the following Robot Assumption 2 where Arm*i*IntoPress (for i = 1,2) denotes the maximal safe extension of the i-th robot arm into the press (i.e. such that the arm will "not crash" with the press) and 0 represents the value of Arm*i*Ext for which the i-th arm is fully retracted.

Robot Assumption 2 .

$$\mathsf{currPhase} \notin \{\mathsf{UnloadingTable}, \mathsf{LoadingPress}\} \Rightarrow \mathsf{Arm1Ext} = 0$$
$$\mathsf{currPhase} \notin \{\mathsf{UnloadingPress}, \mathsf{LoadingDepBelt}\} \Rightarrow \mathsf{Arm2Ext} = 0$$
$$\mathsf{currPhase} = \mathsf{LoadingPress} \Rightarrow \mathsf{Arm1Ext} \leq \mathsf{Arm1IntoPress}$$
$$\mathsf{currPhase} = \mathsf{UnloadingPress} \Rightarrow \mathsf{Arm2Ext} \leq \mathsf{Arm2IntoPress}$$

This assumption implies that the robot never crashes with the press, i.e. "a robot arm may only rotate in the proximity of the press if the arm is retracted or if the press is in its upper or lower position".

Robot Safety Property 2 . *The robot never crashes with the press.*

Proof. By definition of crash, whenever an arm is retracted, it represents no danger. So we have to consider only those phases when the robot extends one of its arms. By Robot Assumption 2 these phases are UnloadingTable and LoadingPress when the robot can extend the first arm, and UnloadingPress and LoadingDepBelt when the second arm can be extended. We can restrict our analysis to LoadingPress and to UnloadingPress because otherwise, by the given geometrical layout, the arm which may be extended is not in front of the press and therefore by definition cannot crash against it. During the phase (Un)LoadingPress Robot Assumption 2 assures that the relevant arm will not crash by extending too much. The definition of the robot rules guarantees that during this phase no crash can occur due to a wrong position of the press or a wrong movement of the robot or of the press; in fact the robot can enter the phase (Un)LoadingPress only by firing a WAITING rule when it is WaitingIn(Un)LoadPressPos (so that by the Robot Assumption 1 it does not rotate) and when PressReadyFor(Un)Loading holds (so that the press is stopped in the correct (un)loading position and is (un)loaded.)

In order to meet the requirement that the blanks held by the arms don't collide with other blanks on the table or on the deposit belt we specify by the following Arm$i$Assumption when the arms are loaded and unloaded.

Arm 1 Assumption . The first robot arm is loaded from the moment when the robot leaves the UnloadingTable phase (when the table has been unloaded) until the end of its next LoadingPress phase (when the press has been loaded).

Arm 2 Assumption . The second robot arm is loaded from the moment when the robot leaves the UnloadingPress phase (when the press has been unloaded) until the end of its next LoadingDepBelt phase (when the deposit belt has been loaded).

Robot Safety Property 3 . *The loaded first arm is never moved above the loaded table, if the table is in unloading position.*

Proof. By definition of the robot rules, by Robot Assumption 2 , by the given geometrical layout of the robot and by the robot placement assumption, arm 1 can be extended towards the table only during the UnloadingTable phase when (by Arm 1 Assumption) arm 1 is not loaded. When it becomes loaded (by application of a robot ACTION rule), the table becomes unloaded and the robot exits the UnloadingTable phase so that by Robot Assumption 2 the first arm has already been retracted from the table.

Robot Safety Property 4 . *The loaded second arm is never moved over the deposit belt unless there is enough space on the belt to receive a new piece.*

Proof. As above arm 2 can be extended over the deposit belt only in phase LoadingDepBelt . By Arm 2 Assumption it is loaded when it enters that phase by firing a WAITING rule. The guard of these rules guarantees that DepositBeltReadyForLoading is true which by the following Robot Assumption 3 implies that the deposit belt has enough free space to hold a new piece.

Robot Assumption 3 . DepositBeltReadyForLoading gets value true if there is enough space on the belt.

We will see below that the formalization of the deposit belt (which can update the function DepositBeltReadyForLoading) implies this assumption.

Remark. We will see in the deposit belt rules that DepositBeltReadyForLoading is set by the deposit belt (i.e. updated from false to true) only when a piece has reached the end of the belt; it is set to false (by the robot) only when it has value true. This implies that at most two pieces are allowed to be on the deposit belt. This is a consequence of the fact that a) there is no sensor indicating when the belt is free to get a new piece, b) we do not know the dimensions of the blanks nor the motor speed. Therefore we have no information on how much time it takes to free enough space for a new piece on the belt. For the formalization of the robot we are not interested to know how the function DepositBeltReadyForLoading gets the value true, we have only to guarantee that its integrity constraint holds. From the point of view of the robot it would be the same to have a sensor on the belt indicating that there is enough space to put a new piece. This is another simple example of how one can get suggestions from the model to enhance the system hardware with a little change on the control software (here interpreting DepositBeltReadyForLoading as a monitored function). The ASM framework seems to be appropriate to deal with such hardware-software co-design problems in a clean and transparent way because of the possibility it offers to integrate the description of a part of a complex system into the whole model using well defined interfaces.

In order to guarantee that the robot arms drop pieces only on the press or on the deposit belt we have to make appropriate assumptions on our abstract notions of LoadingPress and LoadingDepBelt . According to the informal cell description, the robot arms drop pieces by deactivating a magnet; arm 1 should do this only when pointing towards the unloaded press, in its proper extension, arm 2 should do this only when located over the deposit belt. This is expressed by the following Robot Assumption 4 where the function Arm*i*Magnet describes the state of the magnet on the i-th arm.

Robot Assumption 4 .
Arm1Mag is switched to off only if

currPhase = LoadingPress $\wedge$ Arm1Ext = Arm1IntoPress

Same for Arm2 replacing Press by DepBelt.

Robot Safety Property 5 . *The robot never drops pieces outside safe areas.*

Proof. The safe areas for dropping metal blanks are by definition the press and the deposit belt. By the robot rules we know that arm 1 drops a piece (by deactivating its magnet) only in LoadingPress phase so Robot Assumption 4 implies the claim. Similarly, when arm 2 drops a piece, there must be a preceding execution of a WAITING rule guarded by DepositBeltReadyForLoading being true. By Robot Assumption 3 this implies the claim.

Robot Safety Property 6 . *A new blank will be put on the deposit belt only if there is enough space on the belt.*

Proof. A new blank can be put on the deposit belt only during the phase LoadingDepBelt which can be entered only through firing a WAITING rule with true guard DepositBeltReadyForLoading . By Robot Assumption 3 this implies the claim.

Robot Safety Property 7 . *The robot doesn't put blanks into the press if the press is already loaded.*

Proof. A piece can be put into the press only by an action rule preceded by the application of a waiting rule with guard PressReadyForLoading ; by definition this implies not PressLoaded .

3.4 The Press Ground Model

> *... The task for the press is to forge metal blanks. The press consists of two horizontal plates, with the lower plate being movable along a vertical axis. The press operates by pressing the lower plate against the upper plate. Because the robot arms are placed on different horizontal planes, the press has three positions. In the lower position, the press is unloaded by arm 2, while in the middle position it is loaded by arm 1. The operation of the press is coordinated with the robot arms as follows: 1. Open the press in its lower position and wait until arm 2 has retrieved the metal plate and left the press, 2. Move the lower plate to the middle position and wait until arm 1 has loaded and left the press, 3. Close the press, i.e. forge the metal plate. This processing sequence is carried out cyclically. [Lindner 95]*

The press goes through the cycle of loading, forging and unloading under the control of the robot which loads it with blanks and retrieves forged pieces. Abstracting from the motors this can be formalized by a simple automaton using the monitored function PressLoaded (which as we know already is a controlled function for the robot). PressLoaded signals the press when the operations of loading or unloading have been completed such that the lower press plate can safely move to its next position. The monitored functions BottomPosition , MiddlePosition and TopPosition model the three sensors indicating that the lower press plate has reached the corresponding position.

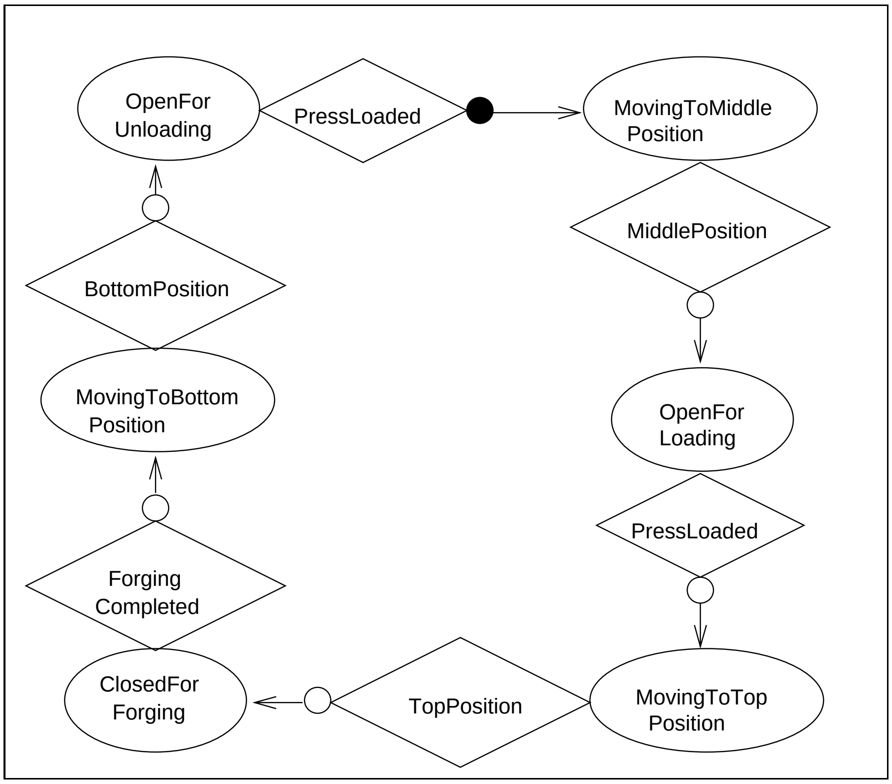
The informal specification provides no means to know when the forging process is completed. We model this through a monitored function ForgingCompleted whose values are determined by the physical environment. (In the toy model presented in [Lindner 95] no forging is actually performed so ForgingCompleted is immediately true once the press has entered the ClosedForForging phase.)

Remark. Our ground model abstraction from the motors realizes an idea which is expressed in the comparative survey, namely that

> *... it would be quite handy to have local microprocessors for each actuator which could help in issuing commands to the machines at a higher level. For instance, instead of switching the motor of the press on and switching it off after the press reached the upper position, then one could simply say "move the press to upper position". This would make the task of writing a control program much easier, and the programs running on the local microprocessors could easier be verified. (see [Lewerentz, Lindner 95b])*

In effect the simplicity of our ground model is partly due to this abstraction from motors which allows us to issue to the agents such abstract moving commands, namely in the form of the updates currPhase(agent):=MovingTo....

Initially the Press is in the OpenForUnloading phase and loaded.



[Press]

WAITING_UNLOAD.
if currPhase= OpenForUnloading and PressLoaded = false
then currPhase := MovingToMiddlePosition

MOVING_TO_MIDDLE.
if currPhase= MovingToMiddlePosition and MiddlePosition
then currPhase := OpenForLoading

WAITING_LOAD.
if currPhase= OpenForLoading and PressLoaded = true
then currPhase := MovingToTopPosition

MOVING_TO_UPPER.
if currPhase= MovingToTopPosition and TopPosition
then currPhase := ClosedForForging

CLOSED.
if currPhase= ClosedForForging and ForgingCompleted
then currPhase := MovingToBottomPosition

MOVING_TO_LOWER.
if currPhase= MovingToBottomPosition and BottomPosition
then currPhase := OpenForUnloading

Press Safety Property 1 . *The press is not moved downward if it is in its bottom position, it is not moved upward if it is in its top position.*

Proof. If the press is moving to its bottom/top position, the movement is stopped by rule MOVING_TO_LOWER and MOVING_TO_UPPER respectively as soon as the press reaches the bottom/top position.

Press Safety Property 2 . *The press does only close when no robot arm is positioned inside it.*

Proof. By definition of the press and robot rules, the press can close only *after* having been open for loading and having been loaded by a robot action. This action, preceding in the given run the closure of the press, has put the robot into a phase (namely MovingToUnloadTablePos) in which (by Robot Assumption 2) both arms are retracted so that in particular none of them is positioned inside the press. By the robot rules, Robot Assumption 1 and Robot Assumption 2 and the robot placement assumption, from that point on in the given run no arm can be positioned again inside the press before the robot enters the phase UnloadingPress. But by definition of the robot rules this can happen only *after* the press has become ready for unloading, i.e. by definition has at least reached the unloading position. Again by definition this means that the press must have entered its OpenForUnloading phase. This proves that there is no run where an arm is positioned inside the press when the press is closed.

3.5 The Deposit Belt Ground Model

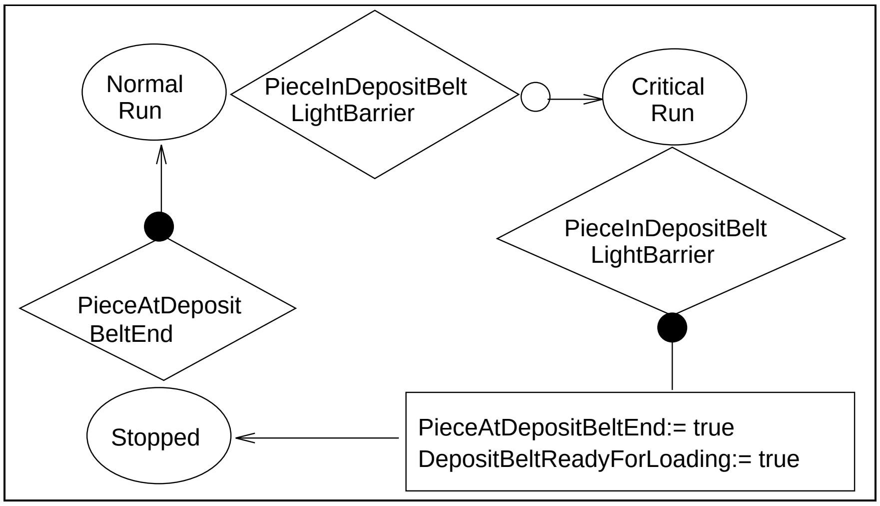
> *... The task of the deposit belt is to transport the work pieces unloaded by the second robot arm to the traveling crane. A photoelectric cell is installed at the end of the belt; it reports when a work piece reaches the end section of the belt. The control program then has to stop the belt. The belt can restart as soon as the traveling crane has picked up the work piece. ... photoelectric cells switch on when a plate intercepts the light ray. Just after the plate has completely passed through it, the light barrier switches off. At this precise moment, the plate is in the correct position to be picked up by the traveling crane ... [Lindner 95]*

The deposit belt is similar to the feed belt but has not to worry about the passage of the piece to the next machine because the traveling crane takes care about that. For a faithful formalization of its only photoelectric sensor, using a 0-ary monitored function PieceInDepositBeltLightBarrier , we need three phases: in NormalRun the deposit belt runs until a work piece intercepts the light barrier at the end section of the deposit belt. At this moment the deposit belt passes to CriticalRun and runs until the plate has completely passed through the light barrier (switching PieceInDepositBeltLightBarrier from true to false). The deposit belt then stops, waiting for being unloaded. A boolean interaction function PieceAtDepositBeltEnd is used to communicate to the traveling crane that a piece has arrived in the unloading area. It is used by the crane to communicate to the deposit belt when the unloading has been completed so that the deposit belt can restart by returning to the NormalRun phase.

The consistency of the updates of PieceAtDepositBeltEnd by the deposit belt and by the traveling crane is guaranteed by the fact that this function is updated to true by the belt only (and only when it is false), and it is updated to false only by the traveling crane (and only when it is true, see the next section).

Besides the function PieceAtDepositBeltEnd we use the interaction function DepositBeltReadyForLoading to signal when there is enough space on the belt to insert a new piece because *"a new blank may only be put on the deposit belt if ... the last one has arrived at the end of the deposit belt"*. Therefore the deposit belt sets DepositBeltReadyForLoading to true whenever a piece has completely passed the light barrier.

The preceding formalization is summarized by the following automaton which cycles through the three phases NormalRun , CriticalRun , Stopped .



For the initialization we stipulate that the current phase is NormalRun and that no pieces are on the belt (i.e.PieceAtDepositBeltEnd = false, DepositBeltReadyForLoading = true).

[Deposit Belt]

DB_NORMAL.
if currPhase= NormalRun and PieceInDepositBeltLightBarrier
then currPhase := CriticalRun

DB_CRITICAL.
if currPhase= CriticalRun and not PieceInDepositBeltLightBarrier
then currPhase := Stopped
    DepositBeltReadyForLoading := true
    PieceAtDepositBeltEnd := true

DB_STOPPED.
if currPhase= Stopped and not PieceAtDepositBeltEnd
then currPhase := NormalRun

The safety requirement for the belt—that it does not drop any metal piece—can be easily proved for our ground model.

Deposit Belt Safety Property . *The deposit belt is stopped when a blank has passed the light barrier at its end and is not re-started until the traveling crane has picked up that blank.*

Proof. Suppose the belt is transporting a piece which arrives at the light barrier at time t_1 and leaves it at a later time $t_2 > t_1$. Then the deposit belt executes the rule DB_NORMAL at time t_1 and DB_CRITICAL at time t_2, stopping the belt and setting PieceAtDepositBeltEnd . The deposit belt can restart only by firing DB_STOPPED which is guarded by PieceAtDepositBeltEnd having become false, signalling that the crane has picked up the piece in question.

Remark. We can now also make Robot Assumption 3 more precise and prove that the deposit belt ASM satisfies it. DepositBeltReadyForLoading becomes true only through the rule DB_CRITICAL which (by the proof of the deposit belt safety property above) can be applied only "when the last blank has arrived at the end of the deposit belt" providing "enough space on the deposit belt to get a new piece".

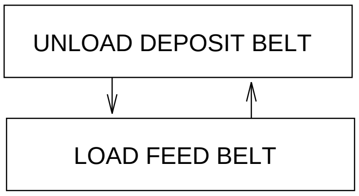
3.6 The Traveling Crane Ground Model

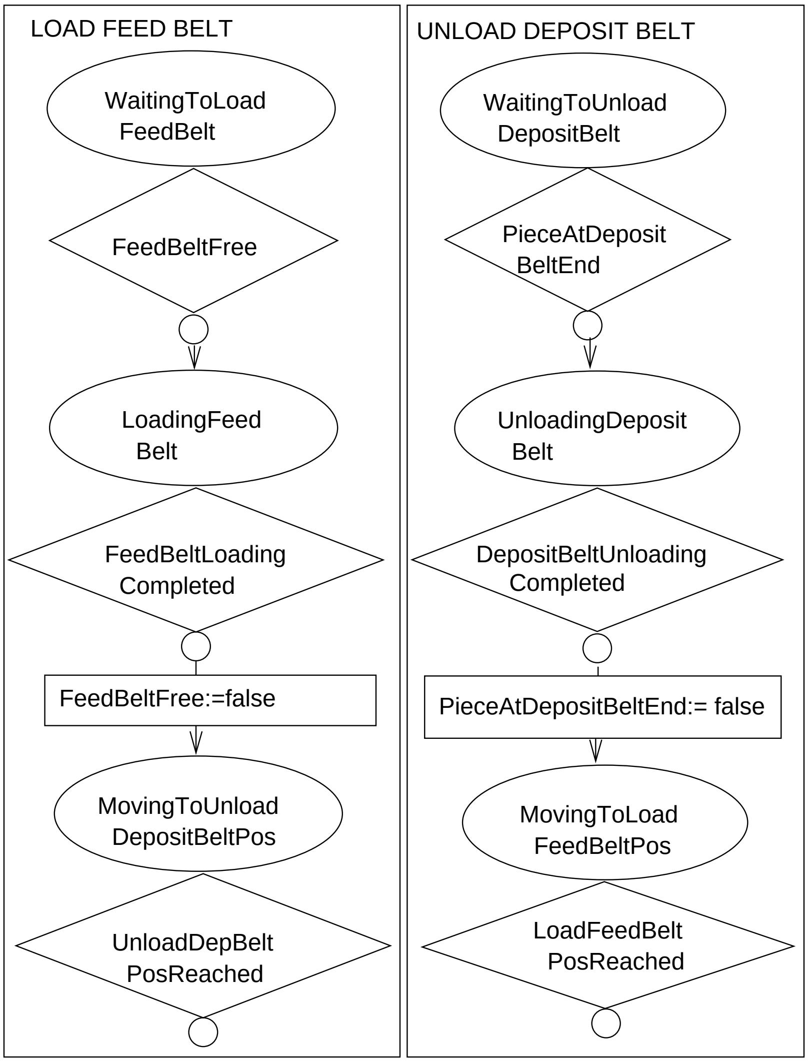
As for the robot ASM, we first specify the Traveling Crane ASM and then prove for it the safety properties required by the task description.

3.6.1 The Specification

> *... The task of the traveling crane consists in picking up metal plates from the deposit belt, moving them to the feed belt and unloading them there. ... The typical operation of the crane is as follows: 1. After the signal from the photoelectric cell indicates that a work-piece has moved into the unloading area on the deposit belt, the gripper positions itself through horizontal and vertical translations over the deposit belt and picks up the metal plate. 2. The gripper transports the metal plate to the feed belt and unloads it there. [Lindner 95]*

Unloading the deposit belt and loading the feed belt can be formalized by two groups of rules which are similar to the corresponding rule groups for the robot and use similar monitored functions *Action*Completed and *Action*PosReached. The crane has to wait until the deposit belt has brought a piece into the unloading area; once a piece has been picked up, the crane switches PieceAtDepositBeltEnd to false and moves to a position where it will wait for the feed belt to be free. Similarly for moving back, after feed belt loading (which sets FeedBeltFree to false), to the position for the next deposit belt unloading.





For the initialization we assume $\mathsf{currPhase} = \mathsf{WaitingToUnloadDepositBelt}$.

[Travelling Crane]

WAITING(DB).
if currPhase= WaitingToUnloadDepositBelt and PieceAtDepositBeltEnd
then currPhase := UnloadingDepositBelt

UNLOADING(DB).
if currPhase= UnloadingDepositBelt and UnloadingDepositBeltCompleted
then currPhase := MovingToLoadFeedBeltPos
    PieceAtDepositBeltEnd := false

MOVING(FB).
if currPhase= MovingToLoadFeedBeltPos and LoadFeedBeltPosReached
then currPhase:= WaitingToLoadFeedBelt

WAITING(FB).
if currPhase= WaitingToLoadFeedBelt and FeedBeltFree
then currPhase := LoadingFeedBelt

LOADING(FB).
if currPhase = LoadingFeedBelt and LoadingFeedBeltCompleted
then currPhase := MovingToUnloadDepositBeltPos
    FeedBeltFree := false

MOVING(DB).
if currPhase= MovingToUnloadDepositBeltPos and UnloadDepositBeltPosReached
then currPhase := WaitingToUnloadDepositBelt

Consistency. We have treated the consistency of the updates of PieceAtDepositBeltEnd in the deposit belt section. The consistency of the FeedBeltFree updates is more complicated because we want our model to work also in the more realistic case that the input is independent from the output and not linked to it by the artificial traveling crane. Therefore FeedBeltFree must be resettable also by the environment and we need an assumption which guarantees that the insertion of pieces by the environment respects the safety requirements. The Insertion Priority Assumption below guarantees that the traveling crane and the environment updates of FeedBeltFree will never be in conflict. The assumption constitutes a basis for a rigorous definition and analysis of various possible hardware/software design decisions with respect to blank insertion.

It remains to show that the updates of FeedBeltFree by the feed belt will never collide with those from the crane or from the environment.

Insertion Priority Assumption . New pieces can be put on the feed belt from outside the cell (whereby FeedBeltFree is set to false) only if there is enough space on the belt (i.e. only if FeedBeltFree = true) and if the travelling crane is not loaded. By definition the travelling crane is not loaded in all the phases which are related to unloading the deposit belt (i.e. MovingToUnloadDepositBeltPos , WaitingToUnloadDepositBelt , UnloadingDepositBelt), otherwise it is loaded.

The crane updates FeedBeltFree only to false, in the rule LOADING(FB), which can be executed only after firing the rule WAITING(FB) which is guarded by FeedBeltFree = true. When the crane excutes WAITING(FB) it is loaded by definition so that (by the Insertion Priority Assumption) between firing WAITING(FB) and firing LOADING(FB) the environment cannot insert a piece (and thereby switch FeedBeltFree to false): thus the crane updates FeedBeltFree only to false and only if it is true. By the Insertion Priority Assumption also the environment updates FeedBeltFree only to false and only if it is true.

We show now that the feed belt can update FeedBeltFree only to true and only when it is false; this implies that its update cannot collide neither with the updates from the crane nor with those from the environment. The feed belt can update FeedBeltFree only to true, namely by the rule FB_NORMAL which can be executed only after PieceInFeedBeltLightBarrier has switched to true—which by our formalization of the sensors presupposes that "a blank has entered the final part of the belt" and therefore that the (initially free) feed belt has previously been loaded with that blank. But whenever a blank is dropped onto the feed belt (either by the crane or by the environment) then FeedBeltFree is switched to false. Therefore FB_NORMAL fires only if FeedBeltFree = false. Notice that initially FeedBeltFree can be updated only by the environment.

3.6.2 The Traveling Crane Safety Properties.

One of the safety requirements for the traveling crane is that the crane gripper has to remain between its two positions over the feed belt and over the deposit belt. One can prove this condition for the traveling crane ground model by assuming for the horizontal movements Travelling Crane Assumption 1 .

Travelling Crane Safety Property . *The traveling crane remains always between the loading position over the feed belt and the unloading position over the deposit belt. If the crane is positioned above the feed belt, it may only move towards the deposit belt, and if it is positioned above the deposit belt, it may only move towards the feed belt.*

Proof. By Travelling Crane Assumption 1 the traveling crane is positioned initially in the unloading position over the deposit belt, can move from there only towards the feed belt and is stopped by firing MOVING(FB) which is executed when the traveling crane has reached the loading position above the feed belt (i.e. LoadFeedBeltPosReached = true). From that position, by the same assumption the traveling crane can move only towards the deposit belt and is stopped by firing MOVING(DB), i.e. when it has reached the unloading position above the deposit belt (i.e. UnloadDepositBeltPosReached = true).

Travelling Crane Assumption 1 . The travelling crane moves towards the feed (deposit) belt only when currPhase = MovingToLoadFeedBeltPos (MovingToUnloadDepositBeltPos). In its Waiting[...] and (Un)Loading[...] phases the travelling crane stays in the corresponding (Un)Loading[...]Pos over the corresponding belt.

The controller is required to avoid also collisions during the vertical gripper movements. Since in the ground model we did not want to reflect the details of how the crane moves the gripper vertically (because it would have forced us to think about implementation features like the different heights of the two belts), we have to make appropriate assumptions on the gripper positions. The height of the gripper is indicated by a monitored function GripperVerticalPos measuring the distance of the gripper from the crane bridge. The value MinimalGripperVerticalPos indicates the minimal distance that the gripper has to stay away from the traveling crane bridge to avoid clashing with the crane support. SafeDistanceFromFeedBelt indicates the maximal distance from the crane bridge in which the gripper can move towards the feed belt without colliding with it, SafeDistanceFromDepBelt is the corresponding value for the deposit belt. The values OnFeedBelt and OnDepositBelt indicate respectively the right gripper height to put a piece on the feed belt or to pick up a piece from the deposit belt.

We use two boolean-valued monitored functions GripperOverFeedBelt and GripperOverDepositBelt supposed to become true if the gripper is above the feed and the deposit belt respectively. An obvious integrity constraint is that GripperOverFeedBelt and GripperOverDepositBelt exclude each other. In terms of the above functions we state Travelling Crane Assumption 2 which suffices to imply the second traveling crane safety property on the vertical movements of the crane.

Travelling Crane Safety Property 2 . *The gripper never crashes with the belts or the fixed part of the traveling crane, i.e. the gripper does not move downward, if it is in the position required for picking up a work piece from the deposit belt, and it does not move upward beyond a certain limit; the traveling crane does not knock against a belt laterally.*

Proof. By Travelling Crane Assumption 2 the gripper never retracts more than the lower bound MinimalGripperVerticalPos which prevents collisions with the fixed part of the crane. From Travelling Crane Assumption 1 and Travelling Crane Assumption 2 we see that when moving towards or staying over a belt, the gripper remains at a safe distance from the belt, the same if it is above the deposit belt.

Travelling Crane Assumption 2 .

MinimalGripperVerticalPos $\leq$ SafeDistanceFromFeedBelt ,
SafeDistanceFromDepBelt , GripperVerticalPos
GripperOverFeedBelt $\Rightarrow$ GripperVerticalPos $\leq$ OnFeedBelt
Same for DepositBelt.
currPhase= MovingToLoadFeedBeltPos |WaitingToLoadFeedBelt
$\Rightarrow$ GripperVerticalPos $\leq$ SafeDistanceFromFeedBelt
Same for Unload and DepositBelt.

Travelling Crane Assumption 3 . The magnet on the gripper is switched to on only during the phase UnloadingDepositBelt and it is switched to off only if the phase is LoadingFeedBelt and GripperVerticalPos = OnFeedBelt .

Travelling Crane Safety Property 3 . *The crane does not drop pieces outside the safe area over the feed belt, i.e. the magnet of the crane may only be* *deactivated, if its magnet is above the feed belt and sufficiently close to it.*

Proof. Follows immediately from Travelling Crane Assumption 3 and the fact that the crane can drop a piece only in phase LoadingFeedBelt .

Travelling Crane Safety Property 4 . *A new blank is put on the feed belt only if there is enough space on the latter to deposit the blank safely.*

Proof: By the definition of the rules for the traveling crane, a new blank can be put on the feed belt only if FeedBeltFree is true. By the insertion priority assumption, the same condition has to hold when the environment can insert a new blank because the traveling crane is not loaded.

3.7 Liveness and Performance Properties

The informal task specification asks to prove the following liveness property (to be understood under the assumption that each time a blank has been inserted, there exists still another blank which can be inserted if the feed belt becomes free again; see above the discussion of the Last Piece Problem).

Liveness. Every blank inserted into the system (by initialization or via the feed belt) will eventually arrive at the end of the deposit belt (and then be dropped by the crane on the feed belt again) and will have been forged.

There are different ways to prove this *Liveness* property. Most of the solutions presented in [Lewerentz, Lindner 95] succeeded to prove only weaker forms of liveness, like for example that at each moment at least one of the machines composing the cell is not stopped. We establish the required strong liveness property and establish also the maximal performance statement showing that our model indeed copes with the maximal number of pieces that the cell is able to process cyclically. Furthermore we show that under natural assumptions our controller "achieves minimal possible time", declared in the informal task description to be "the best result" on efficiency for the production cell control. Moreover the insertion priority assumption guarantees the additional property mentioned in the informal task description as desirable, namely "that the controller takes care that there are never less than a certain number of work pieces in the system, provided that there are enough blanks available."

The order in which the blanks are transported through the points of the production cell gives us the clue to the proof. The *progress points p*—points which can hold blanks—are ordered by the following cyclic successor function *p+1*: *FeedBeltSource, FeedBeltLight Barrier, Elevating RotaryTable, Robot, Press, DepositBeltSource, DepositBeltEnd, TravelingCrane, FeedBeltSource.*

The guards of the rules by which a blank can progress require that $p + 1$ is "free" in order to receive a blank from its predecessor p. Therefore by the pigeon hole principle the production cell would be in a deadlock if it contains 8 blanks.

Let blank x be the occurrence of a blank which is inserted into the production cell via FeedBeltSource through the x-th insertion (for $x > 0$) or via Press through insertion 0 (imposed by the initialization). In the formulation of the following lemma the progress points before the Press have to be treated separately because by initialization blank 0 is inserted into the press and blank 1 can immediately be put on the feed belt and progress unconditionally to the robot.

Blank Progress Lemma. *For every natural number x, every run of the production cell ground model can be extended to a run where the following holds:*

- *blank $i+2$ reaches p after and only after blank $i+1$ has reached $p+1$ for $p \leq$ Robot;*
- *blank $i+1$ reaches p after and only after blank i has reached $p+1$ for Press $\leq p \leq$ TravelingCrane*

for every progress point p and every $i \leq x$.

Proof of the Liveness Property. We assume for the proof the Blank Progress Lemma which will be proved below. Let blank $x \geq 1$ be inserted in a given run via the feed belt. (Due to the Last Piece Problem blank 0 has no chance to arrive at (the end of) the deposit belt unless blank 1 is inserted). By the Blank Progress Lemma, the given run can be extended to a run where blank x has reached the traveling crane and (due to the rule FB-normal) FeedBeltFree is true so that for some $1 \leq n \leq 7$, blank $x+n$ can be inserted via the feed belt. (The particular value of n depends on the speed of the agents and the environment.) Therefore the traveling crane is loaded with blank x so that by the insertion priority assumption the blank $x+n$ to be inserted next is another occurrence of the blank which after insertion x has been transported through all the progress points and now has reached TravelingCrane.

It remains to prove the Blank Progress Lemma. We first prove a lemma which states the conditions under which each sequential agent progresses. We say that an *agent proceeds from a to b, performing update, if condition* if and only if for every state A which is reachable in GroundCELL from the initial state with *agent* in phase a, the following holds: if GroundCELL reaches from A a state satisfying *condition* with *agent* still in phase a, then *agent* is again enabled and can perform *update* and reach a state B in phase b.

Agent Progress Lemma. *The following local progress conditions hold in the ground model GroundCELL:*

FB
- *The Feed Belt proceeds from NormalRun to Stopped or CriticalRun, setting FeedBeltFree, if FeedBeltFree = false.*
- *The Feed Belt proceeds from Stopped or CriticalRun to NormalRun, setting TableLoaded, if TableReadyForLoading.*
- *Initially FeedBelt is in NormalRun.*

ERT
- *The Elevating Rotary Table proceeds from StoppedInLoadPos to StoppedInUnloadPos, if TableLoaded.*
- *The Elevating Rotary Table proceeds from StoppedInUnloadPos to StoppedInLoadPos, if TableLoaded = false.*
- *Initially the Elevating Rotary Table is ready for loading (i.e. in StoppedInLoadPos and not Loaded).*

Robot
- *For every robot Action, Robot proceeds from WaitingInActionPos to WaitingInAction'Pos if ActionCondition, where Action' is the next action the robot may execute after having performed Action and ActionCondition is the second condition in the guard of the Robot rule for Action.*

– *Initially the robot is in WaitingInUnloadTablePos. The sequence of possible robot actions is UnloadingTable, UnloadingPress, LoadingDepBelt, LoadingPress.*

Press
– *Press proceeds from OpenForUnloading to OpenForLoading if PressLoaded = false.*
– *Press proceeds from OpenForLoading to OpenForUnloading if PressLoaded = true.*
– *Initially the Press is OpenForUnloading and loaded.*

DepBelt
– *Deposit Belt proceeds from NormalRun to Stopped, setting PieceAtDepositBeltEnd, if DepositBeltReadyForLoading = false.*
– *Deposit Belt proceeds from Stopped to NormalRun if PieceAtDepositBeltEnd = false.*
– *Initially the Deposit Belt is in NormalRun and can execute no rule.*

TravCrane
– *Crane proceeds from WaitingToUnloadDepositBelt to WaitingToLoadFeedBelt, resetting PieceAtDepositBelt End, if PieceAtDepositBeltEnd.*
– *Crane proceeds from WaitingToLoadFeedBelt to WaitingToUnloadDepositBelt if FeedBeltFree.*
– *Initially Crane is WaitingToUnloadDepositBelt and cannot apply any rule (because PieceAtDepositBeltEnd = false).*

Proof of the Agent Progress Lemma is by inspection of the rules. The Blank Progress Lemma follows from the Agent Progress Lemma by an induction on the number of *insertion blocks* of (≥ 2 and ≤ 7) blanks inserted consecutively through GroundCELL runs until the first block element is loaded on the traveling crane—to be reinserted via the feed belt as first element of the next insertion block; the block length depends on the agent and environment speed.

The basis of the induction follows from the initialization, the insertion priority assumption and the Agent Progress Lemma; namely at the beginning at least 1 and at most 6 blanks can be inserted successively by the environment, in addition to blank 0 inserted into the press by the initialization.

For the inductive step assume that in the given run of GroundCELL m insertion blocks have taken place. Then by the definition of insertion block the traveling crane is loaded with the first blank of block m so that by the insertion priority assumption this blank will be reinserted as first blank of block $m+1$ once the last blank of block m has reached FeedBeltLightBarrier. The rest of the inductive step follows by repeated applications of the Agent Progress Lemma.

Strong Performance Property. *Depending on the speed of the agents and of the environment, the GroundCELL is able to process the maximal number 7 of pieces.*
Once enough blanks have been inserted by the environment so that 7 blanks are present in the machine, there will never be less than 7 blanks in the system (in case of absence of the traveling crane under the additional assumption of eager rule application and blank insertion, i.e. immediately responding agents and blank inserting operator).
No blank stays longer in (any round of) the production cell than needed because under the assumption of eager rule application the GroundCELL achieves the minimum possible time.

Proof. The above proof of the strong liveness property shows that if the environment succeeds to insert 6 new blanks faster than the agents need to load the traveling crane with blank 0, then GroundCELL is able to process 7 pieces simultaneously and will never process less. With eager rule application each agent and the environment process every blank as soon as it can progress. (Nothing more can be said about the time a blank spends in each component unless we have some specific information on the duration of the differents actions.)

4 The Refined Model

In this section we refine the GroundCELL by defining the movements of the agents as driven by the actuators (electric motors and electromagnets) and as watched by the corresponding additional sensors (switches and potentiometers) which occur in the informal task description. This provides a level of abstraction that offers itself for a direct translation to executable code in the next section. The refined model RefinedCELL can easily be proved to correctly implement GroundCELL. It is remarkable that the specification of the distributed ground model as composition of independent submachines which interact through rigorously defined interfaces yields the possibility to combine components even if they are chosen from different levels of abstraction. The rigorous separation of the "local" functionality from the cooperation of the submachines allows us to prove the correctness of the interface preserving refinements separately for each single component, establishing alltogether the following theorem.

Refinement Theorem . *The RefinedCELL implements the GroundCELL correctly.*

Proof. The refinement of the ground model GroundCELL maintains the interfaces, among the submachines and between them and the environment. Therefore the theorem follows from the correctness lemmas stated and proved below showing that each module (i.e. rule set of an agent) in RefinedCELL is a correct implementation of the corresponding abstract one in GroundCELL.

4.1 The Refined Feed Belt

In the informal task description the feed belt is driven by an electric motor which one can model by a function FeedBeltMot with values in { on, off }. One then needs a 0-ary boolean function Delivering to distinguish between the two phases NormalRun and CriticalRun in which the motor is on. This comes up to refine the currPhase-guards as follows whereby the controllable currPhase-function is linked to the system input for the feed belt motor:

$$\mathsf{currPhase} = \mathsf{NormalRun} \equiv \mathsf{FeedBeltMot} = \mathsf{on} \wedge \neg\, \mathsf{Delivering}$$
$$\mathsf{currPhase} = \mathsf{CriticalRun} \equiv \mathsf{FeedBeltMot} = \mathsf{on} \wedge \mathsf{Delivering}$$
$$\mathsf{currPhase} = \mathsf{Stopped} \equiv \mathsf{FeedBeltMot} = \mathsf{off}$$

It is straightforward to rewrite the rules for the ground model feed belt along these lines: changing from NormalRun to CriticalRun (resp. vice versa) means setting (resp. resetting) Delivering , changing from NormalRun to Stopped (resp. vice versa) means resetting FeedBeltMot (resp. setting FeedBeltMot and Delivering . See appendix B where the details are spelled out.

Feed Belt Refinement Lemma. *The refined feed belt RefFB is a correct implementation of the ground model feed belt GroFB. With respect to the refinement definition the runs of GroFB and of RefFB are isomorphic and semantically equivalent (preserving in particular the Feed Belt Safety Property).*

Proof. By the above refinement definition each rule in GroFB is mapped to the homonymous rule in RefFB which by definition has the same effect.

4.2 The Refined Elevating Rotary Table

We define here the abstract action *moving to (un)load position* in terms of rotation and elevation. In the informal task description two motors—which may be started and stopped independently from each other—rotate the elevating rotary table and move it vertically to pass from the position where it can be loaded by the feed belt to the one where it can be unloaded by the robot and viceversa. We formalize this using two 0-ary functions: TableElevationMot has values in {Up, Down, Idle} and TableRotationMot has values in {Clockwise, CounterClockwise, Idle}. The two sensors (switches) which indicate that the table is in its upper or lower position are modeled with two 0-ary boolean functions TopPosition and BottomPosition (which are parameterized by the agent ERT in order to distinguish them from the corresponding functions of the Press). For the potentiometer measuring the angle of rotation of the table to indicate "how far the table has rotated", the controller needs two values, formalized by boolean functions MaxRotation and MinRotation which—following the given geometrical layout—have value true if the table is rotated to the position near the feed belt or to the position where the robot can correctly get a piece respectively.

This yields the following definitions for the currPhase guards in the waiting rules of the elevating rotary table ground model:

$$
\begin{aligned}
&\mathsf{currPhase} = \mathsf{StoppedInLoadPosition} \equiv\\
&\quad (\mathsf{BottomPosition} \wedge \mathsf{MinRotation}\\
&\quad \wedge (\mathsf{TableElevationMot} = \mathsf{Idle}) \wedge (\mathsf{TableRotationMot} = \mathsf{Idle}))\\
&\mathsf{currPhase} = \mathsf{StoppedInUnloadPosition} \equiv\\
&\quad (\mathsf{TopPosition} \wedge \mathsf{MaxRotation}\\
&\quad \wedge (\mathsf{TableElevationMot} = \mathsf{Idle}) \wedge (\mathsf{TableRotationMot} = \mathsf{Idle}))
\end{aligned}
$$

Similarly currPhase = MovingToUnloadPosition is refined to TableElevationMot = Up or TableRotationMot =clockwise and the same for MovingToLoadPosition with Up, Clockwise replaced by Down, counterClockwise. The disjunction appearing in this definition reflects that in the refined model for the elevating rotary table, moving is not anymore an atomic action, but is decomposed into two independent actions. Correspondingly each moving rule in the elevating rotary table ground model is refined by two rules for stopping separately each of the two motors. It is straightforward to refine the initialization condition and the rules along this definition of phase refinement where the start of the two moving actions is triggered by corresponding updates in the refined WAITING rules (see the appendix).

Elevating Rotary Table Refinement Lemma. *Runs of the refined elevating rotary table and of the elevating rotary table ground model correspond to each other via homonymous rules and are semantically equivalent.*

Proof. Follows by induction on runs from the fact that corresponding runs

(WAITING_LOAD , MOVING_UNLOAD , WAITING_UNLOAD , MOVING_LOAD)

(WAITING_LOAD , MOVING_UNLOAD.a|b , WAITING_UNLOAD , MOVING_LOAD.a|b)

of the two ASMs are equivalent; "|" denotes rule execution in any order.

At this level of abstraction it is appropriate and easy to prove the safety property required by the informal task description for the elevating rotary table.

Table Safety Property. *The refined model RefERT for the elevating rotary table never moves over its vertical limits and does not rotate clockwise (resp. counterclockwise) if it is in the position required for transfering blanks to the robot (resp. for receiving blanks from the feed belt).*

Proof. We have to show that the table does never move over the limits represented by the maximal and minimal elevation/rotation. The two stop rules MOVING_(UN)LOAD.(a|b), executed while the table is moving to the position where it can be (un)loaded, guarantee this.

4.3 The Refined Robot

In the informal task description the robot has one motor for extending and retracting each of its arms to reach the table, the press or the deposit belt which are located at different distances from the rotating center of the robot where the two arms are fixed. These motors are formalized by controlled functions Arm1Mot and Arm2Mot with values in { extend, retract, idle }. Two monitored functions Arm1Ext and Arm2Ext model the sensors indicating how much the arms are extended. The intended meaning of their few significant values will be clear from the chosen names (e.g. Arm1IntoPress is the value of Arm1Ext for which the first robot arm is in the position to put a piece into the press).

A controlled function RobotRotationMot models the action of the motor which drives the rotation of the robot base. A monitored function Angle models the sensor indicating the rotation angle for which we need the four values identifying the four positions at which the robot has to stop the rotation to let the arms retrieve or put pieces: Arm1ToTable , Arm2ToPress , Arm2ToDepBelt and Arm1ToPress .Two functions Arm1Mag and Arm2Mag assume the value on if and only if the magnet of the respective arm is switched on.

For these functions we will prove below the assumptions made for the safety proofs in the ground model robot GroR.

In terms of these functions it is easy to define the guards appearing in the ground model for the WaitingIn[...]Pos and the MovingTo[...]Pos phases. The following definition reflects the order of the robot actions which is suggested in the informal task description. WaitingIn[...]Pos means that the rotation Angle has reached the right value for the given Pos(ition) with all motors idle, the arms retracted and the magnets in the correct state. MovingTo[...]Pos differs from Waiting by the rotation motor being on and rotating the robot in the right direction within the corresponding rotation interval. Notationally we speak about intervals for the angle values although (through the discretization assumed for the controller) only a finite number of the real values are relevant.

currPhase= WaitingIn(UnloadTable|UnloadPress|LoadDepBelt|LoadPress)Pos≡
  Angle = Arm1ToTable |Arm2ToPress |Arm2ToDepBelt |Arm1ToPress
   and ArmsRetracted and RobotIdle and Arm1Mag = off|on|on|on
   and Arm2Mag = off|off|on|off

currPhase= MovingTo(UnloadPress|LoadDepBelt|LoadPress|UnloadTable)Pos ≡
  ArmsRetracted and Arm1Mot = idle and Arm2Mot = idle and
  RobotRotationMot = counterClock|counterClock|counterClock|clockwise
  and Arm1Mag = on|on|on|off and Arm2Mag = off|on|off|off and
  Angle ∈ [Arm1ToTable , Arm2ToPress]|
          [Arm2ToPress , Arm2ToDepBelt]|
          [Arm2ToDepBelt , Arm1ToPress]|
          [Arm1ToTable , Arm1ToPress]

RobotIdle ≡ RobotRotationMot = Arm1Mot = Arm2Mot = idle
ArmsRetracted ≡ Arm1Ext = Arm2Ext = retracted

Since the only movement performed by the robot during its MovingTo[...]Pos phases is the rotation of its base, the abstract notion of reaching a position where to stop a moving phase in the ground model can be defined using the values of the angle rotation sensor as follows:

(UnloadPress|LoadDepBelt|LoadPress|UnloadTable)PosReached ≡
  Angle = Arm2ToPress |Arm2ToDepBelt |Arm1ToPress |Arm1ToTable

It is easy to check that with these definitions the moving rules of the ground model are correctly refined by replacing the phase update with stopping the rotation (i.e. robotRotationMot := idle) which defines the movement of the refined robot. It remains to define the robot actions in terms of motors, magnets and sensor. The order of the motor and magnet actions is as follows: a) extend the appropriate arm until it has reached the place to pick up or to drop a piece; b) pick up or drop a piece by switching on or off the magnet of the extended arm (the "proper" action); c) retract the arm until it has become fully retracted (so that the robot can safely rotate to the position of the next action).

Therefore (un)loading is defined as decomposed into three subactions, namely extension, the proper (magnet) action and retraction. For each of these actions it is easy to define the beginning and the completion in terms of sensor and actuator values. (Un)Loading is started by switching on the motor for extending the corresponding arm; when this arm is extended, the proper action takes place by (de)activating the magnet of the arm, followed by switching on the motor for retracting the arm. When the retraction terminates, the robot rotation motor is started. For reasons of simplicity of exposition the following formalization incorporates the magnet switching into the rule for terminating the arm extension.

Extending(Arm1ToTable |Arm2ToPress |Arm2ToDepBelt |Arm1ToPress)≡
  Angle = Arm1ToTable |Arm2ToPress |Arm2ToDepBelt |Arm1ToPress and
  Arm(1|2|2|1)Mot = extend

Extended(OverTable |Arm2IntoPress |OverDepBelt |Arm1IntoPress)≡
  Angle = Arm1ToTable |Arm2ToPress |Arm2ToDepBelt |Arm1ToPress and
  Arm(1|2|2|1)Ext =(OverTable |Arm2IntoPress |OverDepBelt |Arm1IntoPress)
   and Arm(1|2|2|1)Mot = idle

Retracting(Arm1ToTable |Arm2ToPress |Arm2ToDepBelt |Arm1ToPress)≡
  Angle = Arm1ToTable |Arm2ToPress |Arm2ToDepBelt |Arm1ToPress and
  Arm(1|2|2|1)Mot = retract

UnloadingTable |UnloadingPress |LoadingDepBelt |LoadingPress ≡
  Extending(Arm1ToTable |Arm2ToPress |Arm2ToDepBelt |Arm1ToPress) or
  Extended(OverTable |Arm2IntoPress |OverDepBelt |Arm1IntoPress) or
  Retracting(Arm1ToTable |Arm2ToPress |Arm2ToDepBelt |Arm1ToPress)

(UnloadingTable |UnloadingPress |LoadingDepBelt |LoadingPress)Completed ≡
  Retracting(Arm1ToTable |Arm2ToPress |Arm2ToDepBelt |Arm1ToPress)
   and Arm(1|2|2|1)Ext = retracted

For the initialization we assume both arms to be fully retracted (Arm1Ext = Arm2Ext = retracted), the robot rotated such that it points its first arm towards the table (Angle = Arm1ToTable) and completely stopped (i.e. RobotRotationMot = idle and Arm1Mot = Arm2Mot = idle). Both magnets are supposed to be off (Arm1Mag = Arm2Mag = off). This refines correctly the condition imposed on the ground model that the initial phase is WaitingInUnloadTablePos .

The following rules are the result the above refinements where WaitingIn...Pos and MovingTo...Pos denote the definitions given above for currPhase = WaitingIn/MovingTo...Pos.

[Robot]

WAITING.
if WaitingIn(UnloadTable|UnloadPress|LoadDepBelt|LoadPress)Pos and
  (Table|Press|DepositBelt|Press)ReadyFor(Unloading|Unloading|Loading|Loading)
then Arm(1|2|2|1)Mot := extend

ACTION.extension.
if Extending(Arm1ToTable |Arm2ToPress |Arm2ToDepBelt |Arm1ToPress) and
  Arm(1|2|2|1)Ext =(OverTable |Arm2IntoPress |OverDepBelt |Arm1IntoPress)

then Arm(1|2|2|1)Mot := idle
      Arm(1|2|2|1)Mag := on|on|off|off

ACTION.proper.
if Extended(OverTable |Arm2IntoPress |OverDepBelt |Arm1IntoPress)
  and Arm(1|2|2|1)Mag = on|on|off|off
then Arm(1|2|2|1)Mot := retract

ACTION.retraction.
if Retracting(Arm1ToTable |Arm2ToPress |Arm2ToDepBelt |Arm1ToPress)
  and Arm(1|2|2|1)Ext = retracted
then Arm(1|2|2|1)Mot := idle
      RobotRotationMot := counterClock|counterClock|counterClock|clockwise
      TableLoaded|PressLoaded|DepositBeltReadyForLoading|PressLoaded
        :=false|false|false|true

MOVING.
if MovingTo(UnloadPress|LoadDepBelt|LoadPress|UnloadTable)Pos
  and (UnloadPress|LoadDepBelt|LoadPress|UnloadTable)PosReached
then RobotRotationMot := idle

Robot Refinement Lemma. *The refined robot ASM RefR is a correct implementation of the ground robot ASM GroR, i.e. RefR runs and GroR runs correspond to each other via homonymous rules with respect to the refinement definition above and compute the same robot behaviour.*

Proof. The above refinement definition relates the states of the two ASMs in such a way that the corresponding rule guards are equivalent. In mapping homonymous rule groups to each other the refined rules are mapped in the order extension, action, retraction to the corresponding action rule in the ground model. It is a routine exercise to check that via this mapping corresponding runs compute the same result.

It remains to prove that the refined robot ASM satisfies the ground model assumptions 1-4 for the robot and the assumptions for the robot arms.

Assumptions Correctness Lemma. *The robot assumptions 1-4 and the Armi Assumptions of the ground model are satisfied in the refined model.*

Proof. The conditions on the moving and waiting phases and on Angle in Robot Assumption 1 are satisfied in the refined robot model by definition of moving and waiting. It remains to check that during extension and retraction the robot rotation motor is idle. RobotRotationMot initially is idle and remains so through waiting, extension and retraction. Only when the rule ACTION.retraction is executed it becomes clockwise or counterClock for the moving phase and is set back to idle when this phase is terminated by the execution of a MOVING rule.

The robot placement assumption holds in the refined robot model by definition. The appropriate arms begin to extend and are stopped (without exceeding the limit of Robot Assumption 2) when the WAITING and the ACTION.extension rules are executed; they start the retraction when ACTION.proper is executed and stop it at extension 0 by executing ACTION.retraction. This proves that Robot

Assumption 2 holds for the refined robot model.

Robot Assumption 3 on the DepositBeltReadyForLoading function is shown below to hold for the refined model of the deposit belt.

Only the rule ACTION.extension for loading the press can switch off the magnet of the first arm; the rule guard guarantees that the first condition of Robot Assumption 4 is true. Similarly only the rule ACTION.extension for loading the deposit belt can switch off the magnet of the second arm; the rule guard guarantees that the second condition of Robot Assumption 4 is true. Therefore Robot Assumption 4 holds for the refined robot model.

Following the rules of the magnets through a full robot cycle (from waiting in unload table position through press and deposit belt loading to press and table unloading) we can see that Arm 1 Assumption and Arm 2 Assumption are satisfied by RefR. Initially both magnets are switched off so that none of the two arms is loaded. They can be loaded only through an application of an ACTION.etension rule (whereby at the same time the table and the press respectively are unloaded) and remain loaded until the corresponding next execution of an ACTION.extension rule (whereby at the same time the press and the deposit belt respectively are loaded).

4.4 The Refined Press

The press is powered by one motor modeled by a function PressMot with values in { up, down, idle } . We can then define the phases of the press as follows:

currPhase = OpenForUnloading ≡ (BottomPosition and PressMot = idle)
currPhase = MovingToMiddlePosition ≡ (notPressLoaded and PressMot = up)
currPhase = OpenForLoading ≡ (MiddlePosition and PressMot = idle)
currPhase = MovingToTopPosition ≡ (PressLoaded and PressMot = up)
currPhase = ClosedForForging ≡ (TopPosition and PressMot = idle)
currPhase = MovingToBottomPosition ≡ (PressMot = down)

This definition clarifies how the ground model press rules are refined, namely by replacing the phase updates with updates of PressMot; starting to move to the Middle/Top/Bottom position becomes setting the press motor to up/up/down respectively, switching to opening or closing becomes resetting PressMot to idle. The refined Press rules are spelled out in detail in the appendix. We remind the reader that we intend the two functions Bottom/TopPosition as parameterized by the agent Press in order to distinguish them from the corresponding functions of the elevating rotary table.

Press Refinement Lemma. *The refined press is a correct implementation of the ground model press. Their runs are in one-to-one correspondence via the refinement definition and model the same press behaviour.*

Proof. Homonymous rules have the same effect.

4.5 The Refined Deposit Belt

The Deposit Belt is refined similarly to the feed belt using a motor function DepBeltMot with ("boolean") values in { on, off } and a boolean function

Critical to distinguish between the normal and the critical belt run phases. The refinement is defined as follows:

currPhase = NormalRun ≡ (DepBeltMot = on and not Critical)
currPhase = CriticalRun ≡ (DepBeltMot = on and Critical)
currPhase = Stopped ≡ (DepBeltMot = off)

The corresponding rule refinement is obtained by replacing the phase updates to CriticalRun, Stopped and NormalRun with updates of Critical to true, of DepBeltMot to off, of DepBeltMot to on and of Critical to false respectively (see the appendix where the rules are spelled out in detail).

DB Refinement Correctness Lemma. *The refined deposit belt is a correct implementation of the ground model deposit belt. Their runs are in one-to-one correspondence via the refinement definition and model the same table behaviour.*

Proof. Via the refinement homonymous rules have the same effect.
Since no specific assumption has been made for the proof of the deposit belt safety properties for the ground model, that proof is preserved by the refinement.

4.6 The Refined Traveling Crane

The traveling crane is moved by two motors whose action is modeled by two controlled functions CraneVerticalMot (with gripper movement values up, down, idle) and CraneHorizontalMot (indicating whether the crane is moving horizontally (towards the feed belt or the deposit belt) or whether it is stopped). For the sensors we use the monitored functions introduced already for proving the traveling crane safety properties in the ground model. In terms of these functions it is easy to define the waiting phases of the traveling crane as follows:

currPhase =WaitingToUnloadDepositBelt ≡
  GripperOverDepositBelt and GripperVerticalPos = OnDepositBelt and
  CraneHorizontalMot = idle and CraneVerticalMot = idle and
  CraneMagnet = off

currPhase = WaitingToLoadFeedBelt ≡
  GripperOverFeedBelt and GripperVerticalPos = SafeDistanceFromFeedBelt and
  CraneHorizontalMot = idle and CraneVerticalMot = idle and
  CraneMagnet = on

For defining the moving phases we use that from the geometric placement of the two belts and of the crane the following holds:

0= MinimalGripperVerticalPos ≤ OnFeedBelt ≤ OnDepositBelt .

Therefore we assume without loss of generality the following:

SafeDistanceFromFeedBelt = MinimalGripperVerticalPos
SafeDistanceFromDepBelt = OnDepositBelt .

The informal task description contains the information that the feed belt is higher than the deposit belt so that the traveling crane can safely move from the feed belt to the deposit belt with the gripper still in feed belt position, followed by moving the gripper down to the safe deposit belt position. (We assume that the unloaded gripper cannot collide with a piece that eventually will arrive on the deposit belt because it stays at a sufficient height above the belt.) This explains the following definitions for the moving phases of the ground model in the refined model:

MovingToLoadFeedBeltPos ≡
  GripperVerticalPos = SafeDistanceFromFeedBelt and CraneVerticalMot = idle
   and CraneHorizontalMot = toFeedBelt

MovingToUnloadDepositBeltPos ≡
  (CraneHorizontalMot = toDepositBelt and
   GripperVerticalPos = OnFeedBelt and CraneVerticalMot = idle)
      or
  (CraneHorizontalMot = idle and GripperOverDepositBelt and
   CraneVerticalMot = down)

The (un)loading actions of the traveling crane consist of a magnet and a gripper action. To unload the deposit belt the crane has to switch on the magnet on the gripper and then to retract the gripper to a safe distance for moving horizontally to the feed belt. To load the feed belt the crane has to extend the gripper until it has reached a safe distance from the belt and then to switch off the magnet. This leads to the following definition of the ground model (un)loading phases in the refined traveling crane model:

UnloadingDepositBelt ≡
  GripperOverDepositBelt and CraneHorizontalMot = idle and CraneMagnet and
  ((CraneVerticalMot = idle and GripperVerticalPos = OnDepositBelt) or
  (CraneVerticalMot = up))

LoadingFeedBelt ≡
  GripperOverFeedBelt and CraneHorizontalMot = idle and
  ((CraneMagnet and CraneVerticalMot = down) or
    (notCraneMagnet and CraneVerticalMot = idle
      and GripperVerticalPos = OnFeedBelt))

The initialization is refined to the crane being in the position where it can unload the deposit belt with all the motors stopped and the magnet switched off. That implements correctly the initial phase WaitingToUnloadDepositBelt .

Using the preceding definitions it is easy to refine the ground model traveling crane rules. Switching—through an application of the WAITING(DB) rule—to starting the deposit belt unloading becomes turning CraneMagnet on and is followed by applying the two new UNLOADING(DB) rules, namely for starting the (vertical) gripper movement and stopping it upon completion together with switching to moving to the feed belt (by starting the horizontal crane motor). Switching—through an application of the MOVING(FB) rule—to WaitingToLoadFeedBelt becomes stopping the horizontal crane motor. Starting the feed belt loading at the end of the waiting period—through an application of the WAITING(FB) rule—becomes starting to move the gripper down to the feed belt, followed by an execution of the two feed belt loading rules for proper loading (by switching the crane magnet off) and triggering the horizontal movement back to the deposit belt. The rule MOVING(DB) now splits into two for stopping the horizontal movement with simultaneously starting the vertical gripper movement over the deposit belt, followed eventually by stopping the gripper movement and waiting for unloading the deposit belt. The refined rules are spelled out in full in the appendix.

Traveling Crane Refinement Lemma. *The refined traveling crane RefTC is a correct implementation of the grond model traveling crane GroTC. The runs of RefTC correspond to GroTC runs via groups of homonymous rules and via the refinement definition model the same traveling crane behaviour.*

Proof. Follows from the following mapping $\mathcal{F}$ of homonymous rule groups:

$$\begin{aligned}
\mathcal{F}\ ([\mathsf{WAITING(DB)}]) &= [\mathsf{WAITING(DB)}]\\
\mathcal{F}([\mathsf{UNLOADING(DB).a},\ \mathsf{UNLOADING(DB).b}]) &= [\mathsf{UNLOADING(DB)}]\\
\mathcal{F}([\mathsf{MOVING(FB)}]) &= [\mathsf{MOVING(FB)}]\\
\mathcal{F}([\mathsf{WAITING(FB)}]) &= [\mathsf{WAITING(FB)}]\\
\mathcal{F}([\mathsf{LOADING(FB).a},\ \mathsf{LOADING(FB).b}]) &= [\mathsf{LOADING(FB)}]\\
\mathcal{F}([\mathsf{MOVING(DB).a},\ \mathsf{MOVING(DB).b}]) &= [\mathsf{MOVING(DB)}]
\end{aligned}$$

Remark on optimizations. We could reduce the waiting time of the two belts (for FeedBeltFree and PieceAtDepositBeltEnd getting false) by placing the corresponding interface update of the ground model (un)loading rule into the first instead of the second of the refined rules, together with the crane magnet update. It is a routine exercise to show that under reasonable assumptions on the speed of the devices this optimization is correct, i.e. preserves the interface behaviour (including the safety) of the model.

It remains to prove that the refinement definition satisfies the assumptions made in the ground model for proving the traveling crane safety properties.

Proposition. *The Traveling Crane Assumptions 1-3 of the ground model are satisfied in the refined model.*

Proof.

UnloadingDepositBeltCompleted ≡
  GripperOverDepositBelt and GripperVerticalPos = SafeDistanceFromFeedBelt
  and CraneHorizontalMot = idle and CraneVerticalMot = up and
  CraneMagnet = on

LoadingFeedBeltCompleted $\equiv$
GripperOverFeedBelt and GripperVerticalPos = OnFeedBelt and
CraneHorizontalMot = idle and CraneVerticalMot = idle and
CraneMagnet = off

The traveling crane can start to move towards the feed belt only by applying rule UNLOADING(DB).b whose guards and updates guarantee that MovingToLoadFeedBeltPos is true. The traveling crane can start to move towards the deposit belt only by applying the LOADING(FB).b rule. The guards and updates of that rule guarantee that MovingToUnloadDepositBeltPos is true. This proves Travelling Crane Assumption 1 .

The following condition:

SafeDistanceFromFeedBelt $\geq$ MinimalGripperVerticalPos
SafeDistanceFromDepBelt $\geq$ MinimalGripperVerticalPos

in Travelling Crane Assumption 2 is satisfied by the assumption made above; GripperVerticalPos $\geq$ MinimalGripperVerticalPos is true because the gripper starts moving upward only when the rule UNLOADING(DB).a is fired, this movement is stopped by the execution of UNLOADING(DB).b whose guard guarantees that the MinimalGripperVerticalPos is never passed.

As shown above MovingToLoadFeedBeltPos becomes true when the rule UNLOADING(DB).b is executed, it becomes false (with WaitingToLoadFeedBelt becoming true) by execution of the rule MOVING(FB); when thereafter the rule WAITING(FB) is executed, WaitingToLoadFeedBelt gets value false. When UNLOADING(DB).b is fired, the traveling crane has a SafeDistanceFromFeedBelt ; thereafter it is moved only by firing WAITING(FB). Therefore the refined model satisfies the following condition:

currPhase= MovingToLoadFeedBeltPos |WaitingToLoadFeedBelt
$\Rightarrow$ GripperVerticalPos $\leq$ SafeDistanceFromFeedBelt

Examination of the rules shows that the gripper is never moved to a vertical position lower than the position where it can get a piece from the deposit belt, namely (see above) SafeDistanceFromDepBelt = OnDepositBelt . Therefore also the following is true:

currPhase= MovingToUnloadDepositBeltPos |WaitingToUnloadDepositBelt
$\Rightarrow$ GripperVerticalPos $\leq$ SafeDistanceFromDepBelt

By examining the rules one can also see that when the gripper is over the feed belt (i.e. GripperOverFeedBelt = true), then its downward movement is stopped once it has reached the OnFeedBelt distance from the crane bridge and is not moved further as long as GripperOverFeedBelt = true. Thus also the following condition is true:

GripperOverFeedBelt $\Rightarrow$ GripperVerticalPos $\leq$ OnFeedBelt .

A similar argument shows that the gripper is never moved below the OnDepositBelt position when GripperOverDepositBelt = true. Therefore Travelling Crane Assumption 2 is true in the refined model.

For the proof of Travelling Crane Assumption 3 it suffices to observe that the gripper magnet is switched on only by applying rule WAITING(DB) and keeps this value during the deposit belt unloading until it is switched off by the rule LOADING(FB).a (in the loading feed belt phase when GripperVerticalPos = OnFeedBelt).

5 The Refinement to C^{++} Code

In this section we introduce the refinement of the ASM RefinedCELL to a C^{++} program which we have shown by extensive experimentation to control successfully the simulation of the production cell (proposed in [Brauer, Lindner 95] as a validation environment for the specifications and the implementations). Based upon the ASM definition of the semantics of C^{++} (see [Wallace 95]) one also has the possibility to *prove*, by a detailed mathematical argument, that RefinedCELL is correctly implemented by the C^{++} program. The reader will see that writing down the detailed proof could be a long, but not a difficult task because the coding is guided by the structure of the ASM model and because the refinement relation from RefinedCELL to the C^{++} program is simple. The structural similarity between the two is so close that transforming RefinedCELL into the code was to a great extent a matter of (mechanizable) cut and paste. To some it may come as a surprise that one can combine the advantages of using an efficient and rich language like C^{++} and of a clean, module and interface oriented, structured design method which provides the possibility of turning design reasoning into rigorous correctness arguments.

We explain here only the overall structure of the translation to C^{++} code and refer for the details to [Mearelli 97] which contains, for a complete documentation of this last refinement step, the entire executable controller.

5.1 The Structure of the Control Program

The informal task desciption (see [Lewerentz, Lindner 95a]) requires that the control

> *... program ... reads sensor values from UNIX* stdin *and writes its control commands to* stdout *according to standardized ASCII protocol.*

The controller can use two kinds of commands, namely *commands to get status information from the simulation* and *commands for switching the devices on and off* (together with a command **blank_add** which can be used by the controller to simulate the actions of an operator inserting pieces from outside the cell). The status information comes as a 15-element vector indicating the sensor values and the errors which occurred since the last status request.

Therefore it is natural to implement each submachine of the RefinedCELL by a module and to add a module to handle the errors which are reported by the simulation. The most important implementation decision is about how to sequentialize the distributed RefinedCELL in a semantics preserving and provably correct way. We have to decide upon the order in which the controller asks the simulation environment for new status information, forwards it to the seven modules and executes them (thereby implementing the reaction of the agents to the changes in their environment). Such scheduling principles are easily dealt with at a high level of abstraction; see [Börger,Durdanovic 96] where this idea has been developed for the (correctness proof of the) sequentialization of Occam programs as part of their implementation by Transputer code. We abstain here from further developing this idea because for the production cell the sequentialization and the proof of its correctness are a routine exercise if one follows the order applied by the simulator.

This results in the main program structured as a loop where the controller first asks the simulation for new status information. Then each machine—declared to be an object from the corresponding class—gets its updated sensors readings from the standard input, in the order specified by the status vector in the simulator (see [Brauer, Lindner 95]), and eventually reacts updating its internal status and outputting the required commands. Finally the errors provided through the last vector element are taken by the controller from the input stream and processed.

This yields the following C++ code for the main program control.cc where the seven classes represent the seven program modules and control.h contains the class definitions and the definition of the modules AskNewStatus() (for the interface of the controller to the simulator) and Oper() which implements the behaviour of an operator continuously trying to insert new pieces on the feed belt under the regime of the Insertion Priority Assumption .

```
#include "control.h"

cElist        Errors;
cFeedBelt     FeedBelt;
cElevRotTable ElevatingRotaryTable;
cRobot        Robot;
cPress        Press;
cDepositBelt  DepositBelt;
cTravCrane    TravelingCrane;

int main()
{ loop
    { AskNewStatus();
      cin>> Press
         >> Robot
         >> ElevatingRotaryTable
         >> TravelingCrane
         >> FeedBelt
         >> DepositBelt
         >> Errors;
      Oper();
    };

}
```

The initialization of the cell is performed by the constructors of the classes. The initialization of the C++ program differs slightly from the initialization of the ASM models. The differences are mainly motivated by the desire to insert initially all blanks through the feed belt (avoiding the preloading of the press) and concern only the robot, the press and the traveling crane. We skip here the easy exercise in standard preprocessing techniques to add a preprocessor to GroundCELL and RefinedCELL which preserves the required safety assumptions and brings the model into the initial state assumed above for our ASM models.

For the possible delay between the moment when the sensors values are read and the moment when the controller can react we need an assumption to guarantee that our controller is *fast enough* to control the simulation correctly.

Simulation Assumption. Each time a sensor of the cell gets a new value, a) the controller has enough time to read the new value and to react by emitting the required commands, b) the simulation can execute the commands *before* any changes to the value of the sensor are **detected**.

We have verified during the many hours of test runs that this assumption is valid for the simulation and our controller.

5.2 From Component ASMs To C++ Modules

In order to uniformly reflect also the structure of the ASM component machines of RefinedCELL in the structure of the corresponding C^{++} class, we derive each cell-component class from an abstract class cModule which has abstract member functions Where and Rules implementing for each machines its private *macros* and its rules respectively. Notationally we switch here to the common practice to define the macros before the instructions where they are used.

```
class cModule { protected:
                      virtual void Where() = 0;
                      virtual void Rules() = 0;  };
```

The classes derived from cModule overload the input operator >> such that it can be used to retrieve from the input stream the new values for the sensors and then to call the Where and Rules functions.

The ASM functions modeling the sensors and the actuators are implemented by two class templates cSensor and cActuator; they are templates because the various sensors or actuators have different value ranges or commands. Thus in each class describing a submachine we have an object of the class cSensor for each sensor and one of the class cActuator for each actuator of that machine.

As example we include the feed belt class code. The reader will recognize the various functions and macros from the refined feed belt ASM which have now become variables and conditions. It is a routine exercise to show that this code implements the refined feed belt ASM correctly. Remember that the constructor intializes the belt status functions.

```
class cFeedBelt : public cModule
{ bool Delivering;
  bool NormalRun;
  bool CriticalRun;
  bool Stopped;
  bool TableReadyForLoading;
```

```
  // The Sensor
  cSensor<bool> PieceInFeedBeltLightBarrier;

  // The Actuator
  cActuator<OnOff> FeedBeltMot;

  void Where()
    { NormalRun = (FeedBeltMot == on)&&(!Delivering);
      CriticalRun = (FeedBeltMot == on)&&Delivering;
      Stopped = FeedBeltMot == off;
      TableReadyForLoading = TableInLoadPosition &&
                                   (!TableLoaded);
    };

  void Rules()
    { if (NormalRun && PieceInFeedBeltLightBarrier())
      {FeedBeltFree = true;
       if (TableReadyForLoading)
         { Delivering = true;}
       else { FeedBeltMot = off;};};

      if (CriticalRun && (! PieceInFeedBeltLightBarrier()))
        {  Delivering = false;
           TableLoaded = true;};

      if (Stopped && TableReadyForLoading)
          { FeedBeltMot = on;
            Delivering = true;};
    };

 public:
   cFeedBelt()
    : PieceInFeedBeltLightBarrier(false), FeedBeltMot(FB_com, 2)
    { FeedBeltMot = on;
      Delivering = false;
      NormalRun = true;
      CriticalRun = false;
      Stopped = false;
      TableReadyForLoading = TableInLoadPosition &&
                                   (!TableLoaded);
    };

  friend istream& operator>>(istream& is, cFeedBelt& fb)
    { is>> fb.PieceInFeedBeltLightBarrier;
      fb.Where();
      fb.Rules();
      return is;
     };
};
```

The reader can find the complete refinement to C^{++} modules in [Mearelli 97].

6 Evaluation and Conclusions

We answer here the questions posed in [Lindner 95] to evaluate the proposed problem solution and compare from the methodological point of view our solution to the solutions in [Lewerentz, Lindner 95].

6.1 Answers to the Evaluation Questions

We have proved (for the formal requirement specification model and for the refined model) *all* the required safety, liveness and performance properties and have established without difficulties the maximal throughput of the system (a property which the informal task description declares to be "very difficult to prove"). Our proofs are simple because we could exploit for them the abstraction features built into the ASM notion which allowed us to separate the proof obligations for each submachine and to establish the interaction properties on the basis of transparent (precisely defined) interfaces. All the assumptions concerning the cell behavior or the architecture appear explicitly in our specification and are documented. Some of the abstract assumptions made in the ground model have guided the refinement process and the code development.

We have tested our C^{++} code extensively and with success against the FZI simulator in Karlsruhe; we didn't experience a single failure in controlling the simulator under the required safety, liveness and performance conditions.

The reader will judge whether he shares our conviction that both the ground model and the refined model are a) as short and simple as the system to be built does permit, and b) can be understood by every experienced programmer without any familiarity with the ASM method—it suffices to use and read the ASMs as *pseudocode over abstract data*. Since ASMs offer the possibility of a satisfactory solution for the ground model problem, the ASM approach can be used as a candidate "for intuitive and adequate formalisms that allow for building models which closely simulate reality and support discussions with the customer" (see [Lewerentz, Lindner 95b]). The process of stepwise development, and the use of explicit assumptions, provide a complete documentation of the system which makes it easy to change the controller and to reuse proofs—the use of abstraction for components and for interfaces in the hierarchical ASM design approach guarantees maximal design flexibility, re-usability and extendability through easy adaptations of ASMs to evolving requirement specifications; this is particularly important due to the experience of various formal method case studies that "even problems of modest real world complexity are just within the current limits of available automatic proof tools and model-checkers" (see [Lewerentz, Lindner 95b]; see [Abrial,Börger,Langmaack 96] where the same experience is reported). Due to our abstract (object oriented and modular) approach, typically a change in one part of the cell does *not* invalidate all proofs; for example if one makes local changes to the model while leaving the interfaces among the modules unchanged, we just need to update the proofs relative to that module.

Along the way we have illustrated through simple examples that the ASM method fits well as description vehicle for Hardware/Software Co-Design, due to the abstraction mechanisms offered by ASMs. This answers positively the question whether it is possible to draw from our solution conclusions on how the hardware design of the production cell could be improved. In this connection it is interesting to observe that our ASM cell models can be easily extended to reflect the additional hardware (another press, more sensors for the two belts, for the table and the press, etc.) and the failure situations introduced in a recent fault-tolerant extension of the production cell (see [Lötzbeyer 96a]). Also the real-time properties proposed in another extension (see [Lötzbeyer 96b]) can be incorporated into our models using techniques from [Börger et al. 95, Gurevich, Huggins 96]). The reader may enjoy to verify this claim.

6.2 Comparison to Other Solutions

[Lewerentz, Lindner 95b] contains a detailed comparative survey of the solutions in [Lewerentz, Lindner 95] so that we limit ourselves to mention only the salient features which distinguish them from the ASM solution proposed here.

Our ASM models are considerably simpler than the numerous finite state machine or transition system based models in [Lewerentz, Lindner 95]. To mention just one concrete example we suggest to compare the ASM models (and their graphical visualization) for the elevating rotary table with the corresponding statechart specification (see [Damm et al. 95, pages 135-138]); the authors admit that "the conditions of the transitions are very complex" so that they could not be layed down graphically, missing already for the specification of such a simple device one of the declared "key features of the presented approach", namely "the use of graphical specification techniques". The simplicity of our ASM models (and of their graphical definition) is due to the systematic use of abstraction and refinements which allows us to remain close to the informal task description (the ground model problem), to obtain simple proofs for the required system properties and to provide models for structured and easily modifiable code—and this also in cases which are more challenging than the finite automaton like production cell.

The possibility offered by the ASM approach to support a theoretically sound but nevertheless simple and practical integration of different methods for different concerns—code development, mathematical or machine verification and documentation—is a distinctive feature of the method (see [Beierle et al. 96] for its application to the Steam Boiler Control case study). Just one comparison must suffice here as example to illustrate this claim. The solution which uses the FOCUS method is explained through the really rather trivial elevating rotary table as running example (sic) and produces a Concurrent ML program implementing an abstract stream based specification; it is admitted however that "since so far Concurrent ML has no denotational semantics", it is not possible to prove the correctness of this functional program which the authors suggest to transform further into a procedural program (presumably in a language with rigorously—functionally?—defined semantics as reference for proving the correctness of this additional transformation). Why not use right away the operational semantics for Concurrent ML (see [Nielson 96])? The problem the FOCUS method has with applications derives from its commitment to the denotational cause: *The classical denotational paradigm ... has some definite limitations. Firstly, fine-structural features of computation, such as sequentiality, computational complexity, and optimality of reduction strategies, have either not been captured at all* *denotationally, or not in a fully satisfactory fashion. Morevore, once languages with features beyond the purely functional are considered, the appropriateness of modeling programs by functions is increasingly open to question. Neither concurrency nor 'advanced' imperative features have been captured denotationally in a fully convincing fashion* (see [Abramsky 97]). In comparison, the ASM concept is operational but abstract; it has a simple theoretical foundation, described in [Gurevich 95] starting from scratch and understandable by the working computer scientist without any theoretical prerequisites; the method based on this concept reflects and supports current practice and therefore can be applied *as is* at each level of the software development life cycle. For our solution we can base a correctness proof of the C^{++} code with respect to the refined ASM model on the rigorous ASM definition of the semantics of C^{++} given in [Wallace 95].

Some solutions have put a particular emphasis on exhibiting how to support modifiability and reuse during the design process. For example in the SDL specification (see [Heinkel, Lindner 95]) the belts have been designed as instantiations or modifications of a "general" belt, similar uniformisations have been proposed for the vertical and horizontal moves (of the robot and the crane) and for the rotation (of the robot and the table). All these reusability features spring out rather naturally from the corresponding abstractions in our ASM models and are visible in the symmetry of the rules for the feed and the deposit belt, in the symmetry of the four groups of waiting/action/moving rules of the robot, etc. In addition we can reuse not only the specifications but also the proofs to establish the desired system properties.

Some solutions have separated the specification of local state transitions from imposing the restrictions due to the safety conditions or the specification of the components of the system from the specification of the controller. In contrast our ground model exposes explicitly—but abstractly—all the interfaces and the conditions which determine the functionality of the components as part of the desired system behaviour; the ground model defines the control of the system in such a way that the safety conditions can be shown to hold under explicitly stated assumptions on the abstractions (which have to be preserved by the refinements). As a result we have simpler specifications, simpler proofs and a more directly process oriented model which can be made executable as a prototype and—being complete—can serve for checking the correctness of the formalization through discussion with the customer. Since the ASM models can always be made "complete" in this sense, it is always possible (but not mandatory) to make them executable, at any desired level of abstraction. In the RAISE approach, which shares many of the properties of the ASM method but comes with a more restricted language and more restricted proof possibilities, the intermediate models are only mentally simulatable.

A major difference of our production cell models with respect to others is the strict separation of safety from error concerns. We formalize the behaviour of the components only so far as it is relevant for the desired regular system behaviour. The definition of the controller is geared to guarantee this intended safe system behaviour. To deal with definition, detection and handling of physical errors—say that a blank falls down from a belt or that a magnet becomes defective—is relegated to a separate error handling ASM. This approach has been used with success for integrating the Prolog error handling mechanism into the ASM definition of the ISO Prolog standard semantics (see [Börger, Rosenzweig 94]).

Our ASM models come with some useful object oriented features resembling those reported in the object oriented solutions in [Lewerentz, Lindner 95]. Basic object oriented features are inherent in the data and action abstraction mechanism of ASMs and have facilitated the transition from the refined ASM model to structured C^{++} code. This transparent code structure, which is driven by the refinement process and enhances the reusability of design and proof elements, constitutes also a major difference of the ASM solution with respect to the "flat" descriptions using Esterel, Lustre, Signal. In a sense this holds also with respect to the successful model checking solution described in [Nökel, Winkelmann 95]. However, by enriching finite state machines with powerful abstraction mechanisms, ASMs provide a chance to make a powerful symbolic model checker applicable also to systems where the traditional approach brakes down due to the state explosion problem; we will gain both, the advantages offered by decomposing complex systems into smaller systems, at different levels of abstraction, and the advantages of exploiting state-of-the-art tools for machine supported verification and program synthesis.

A major difference with the (temporal logic based) TLT solution concerns the treatment of the synchronization of the actions of the components. The TLT solution starts with an abstract centralized model which is refined by introducing the environment (through instructions for reading sensor values and for setting actuator values) and is then distributed by introducing a scheduling mechanism which forces different components to execute certain instructions simultaneously (for example deliverTable and collectRobot). Our ASM models avoid the somehow artificial grouping of rules concerning more than one agent; the synchronization (read: sequentialization) of specific actions of otherwise independent modules is obtained in the ASM models through appropriate interfaces (in the example: TableLoaded). Also the synchronization events used in [Rischel,Sun 97] to define interfaces avoid any splitting of rules and allow the authors to decompose their program into small, manageable components, closely resembling the structure of our component machine rules. One can map our interface updates to the interface defining event pairs in [Rischel,Sun 97] (for example *begin_t_a1*$\rightarrow$*end_t_a1* corresponds to the update TableLoaded := false in the robot rule whereafter the table can terminate its waiting phase for unloading and proceed to moving to the loading position). The CSP-style formalization is geared to treat synchronization by a uniform communication scheme which abstracts from the explicit formulation of the "interface content" of the action (in the example: that a function TableLoaded is updated by one agent to be read by another agent).

Acknowledgements. We thank Peter Päppinghaus for his help in elaborating the classification of functions which appeared in [Börger 95]. We thank for the valuable criticism received from our referees and from the following colleagues who have read and commented upon the draft versions of this work: Giuseppe Del Castillo, Arnaud Durand, Yuri Gurevich, Paul Joannou, Thomas Lindner, Jacques Loeckx, Cecilia Mascolo, Alfonso Pierantonio, Hans Rischel, Gerhard Schellhorn, Karl Stroetmann, William Sherman, Kirsten Winter. Last but not least the first author thanks the students of his Spring 1997 software engineering course for their attention and critical questions.

References

[Abramsky 97] Abramsky, S.: “Semantics of Interaction”; Pitts, A., Dybjer, P. (eds),“Semantics and Logics of Computation”, Proceedings of the Newton Institute, Cambridge University Press 1997.

[Abrial,Börger,Langmaack 96] Abrial, J.-R., Börger, E., Langmaack, H. (eds.): “ Formal Methods for Industrial Applications (Specifying and Programming the Steam-Boiler Control)”; Springer LNCS State–of–the–Art Survey 1165 (1996), VIII+511.

[Bharadwaj,Heitmeyer 97] Bharadwaj, R., Heitmeyer, C.: “Verifying SCR Requirements Specifications Using State Exploration”; Proc. First ASM SIGPLAN Workshop on Automatic Analysis of Software, Jan. 1997.

[Beierle et al. 96] Beierle, C., Börger, E., Durdanovic, I., Glässer, U., Riccobene, E.:“Refining abstract machine specifications of the steam boiler control to well documented executable code”; Abrial, J.-R., Börger, E., Langmaack, H. (Eds.), Formal Methods for Industrial Applications (Specifying and Programming the Steam-Boiler Control), Springer LNCS State–of–the–Art Survey 1165 (1996), 52-78.

[Börger 94] Börger, E.:“Logic Programming: The Evolving Algebra Approach”; Pehrson, B., Simon, I. (Eds.), IFIP 13th World Computer Congress 1994, Volume I: Technology/Foundations, Elsevier, Amsterdam (1994), 391-395.

[Börger 95] Börger, E.:“Why Use Evolving Algebras for Hardware and Software Engineering?”, Bartosek, M., Staudek, J., Wiedermann, J.(Eds), SOFSEM’95 (22nd Seminar on Current Trends in Theory and Practice of Informatics); Springer LNCS 1012 (1995), 236–271.

[Börger 95a] Börger, E.: “Annotated Bibliography on Evolving Algebras”; Börger, E. (ed.), Specification and Validation Methods, Oxford University Press (1995) 37-51.

[Börger 96] Börger, E.: “ Evolving algebras and Parnas tables”; Ehrig, H., von Henke, F., Meseguer, J., Wirsing, M. (eds), Specification and Semantics, Dagstuhl Seminar Report 1996.

[Börger,Durdanovic 96] Börger, E., Durdanovic, I.: “Correctness of Compiling Occam to Transputer Code”; The Computer Journal 39,1 (1996), 52-92.

[Börger et al. 95] Börger, E. , Gurevich, Y., Rosenzweig, D.: “The Bakery Algorithm: Yet Another Specification and Verification”; Börger, E.(ed), Specification and Validation Methods, Oxford University Press (1995), 231–243.

[Börger, Mazzanti 97] Börger, E., Mazzanti, S.:”A Practical Method for Rigorously Controllable Hardware Design”; Bowen, J.P., Hinchey, M.G., Till, D. (eds), ZUM’97: The Z Formal Specification Notation, Springer LNCS 1212 (1997), 151-187.

[Börger, Rosenzweig 94] Börger, E., Rosenzweig, D.: “The WAM Definition and Compiler Correctness”; Beierle, C., Plümer, L.(Eds.), Logic Programming: Formal Methods and Practical Applications, North-Holland, Series in Computer Science and Artificial Intelligence (1994), 20-90

[Börger, Rosenzweig 94] Börger, E., Rosenzweig, D.: “A Mathematical Definition of Full Prolog”; Science of Computer Programming 24 (1995), 249–286.

[Brauer, Lindner 95] Brauer, A., Lindner, T.: ”Simulation”; Springer LNCS 891 (1995), 273-284

[Damm et al. 95] Damm, W., Hungar, H., Kelb, P., Schlör, R.: “Statecharts”; Springer LNCS 891 (1995), 131-150.

[Del Castillo et al. 96] Del Castillo, G., Durdanovic, I., Glässer, U.: “An evolving algebra abstract machine”; Kleine Büning, H. (ed), Computer Science Logic, Springer LNCS 1092 (1996), 191-214. See ”Available Tools” on http://www.uni-paderborn.de/cs/asm.html.

[Gurevich 95] Gurevich, Y.: "Evolving Algebras 1993: Lipari Guide"; Börger, E. (ed.), Specification and Validation Methods, Oxford University Press (1995) 9-36.

[Gurevich, Huggins 96] Gurevich, Y., Huggins, J.: "The Railroad Crossing Problem: An Experiment with Instantaneous Actions and Immediate Reactions"; Kleine Büning, H. (ed), Computer Science Logic, Springer LNCS 1092 (1996), 266-290

[Hall 97] Hall, A.: "Taking Z Seriously"; Bowen, J.P., Hinchey, M.G., Till, D. (eds), ZUM'97: The Z Formal Specification Notation, Springer LNCS 1212 (1997) 89-91.

[Heinkel, Lindner 95] Heinkel, S., Lindner T. :"SDL"; Springer LNCS 891 (1995), 171-183.

[Heitmeyer 97] Heitmeyer, C.L.: "Formal Methods: A Panacea or Academic Poppycock?"; Bowen, J.P. , Hinchey, M.G. , Till, D. (eds), ZUM'97: The Z Formal Specification Notation, Springer LNCS 1212 (1997), 3-9.

[Lewerentz, Lindner 95] Lewerentz C., Lindner T. (eds.):" Formal Development of Reactive Systems. Case Study "Production Cell""; Springer LNCS 891 (1995).

[Lewerentz, Lindner 95a] Lewerentz, C., Lindner, T.: "Introduction"; Springer LNCS 891 (1995), 3-8.

[Lewerentz, Lindner 95b] Lewerentz C., Lindner T. :" Comparative Survey"; Springer LNCS 891 (1995), 21-54.

[Lindner 95] Lindner. T.: "Task Description"; Springer LNCS 891 (1995), 9-21.

[Lötzbeyer 96a] Lötzbeyer, A.: "Task Description of a Fault-Tolerant Production Cell"; FZI Karlsruhe, Version 1.6. (1996).

[Lötzbeyer 96b] Lötzbeyer, A.: "Task Description of a Flexible Production Cell with Real Time Properties"; FZI Karlsruhe, Version 2.1. (1996).

[Mearelli 97] Mearelli, L.:"Refining an ASM Specification for the Production Cell to C^{++} Code"; J.UCS (Journal for Universal Computer Science), this volume.

[Nielson 96] Nielson, F. (Ed.): "ML With Concurrency"; Springer Monographs in Computer Science, 1996.

[Nökel, Winkelmann 95] Nökel, K., Winkelmann, K.: CSL.Controller Synthesis and Verification: A Case Study"; Lewerentz, C., Lindner, T. (eds.), Formal Development of Reactive Systems. Case Study "Production Cell"; Springer LNCS 891 (1995), 55-74

[Päppinghaus 97] Päppinghaus, P.: "Industrial Use of ASMs for System Documentation"; Jähnichen, S., Loeckx, J., Smith, D.R., Wirsing, M. (eds.), Dagstuhl Seminar on Logic for System Engineering, Dagstuhl Seminar Report (1997).

[Parnas, Madey 95] Parnas, D.L., Madey, J.: "Functional documents for computer systems"; Science of Computer Programming 25 (1995), 41-62.

[Pusch 96] Pusch, C.:" Verification of Compiler Correctness for the WAM"; von Wright, J., Grundy, J., Harrison, J. (eds.), Theorem Proving in Higher Order Logics, TPHOLs'96, Turku, Springer LNCS 1125 (1996.), 347-362.

[Rischel,Sun 97] Rischel, H., Sun, H.: "Design and Prototyping of Real-Time systems Using CSP and CML"; 9th EUROMICRO Workshop on Real-Time Systems, Toledo June 11-13 (1997).

[Schellhorn, Ahrendt 97] Schellhorn, G., Ahrendt, W.: "Reasoning about Abstract State Machines: The WAM Case Study"; J.UCS (Journal for Universal Computer Science) Special ASM Issue (Part I) 3,4 (1997), 377-413.

[Wallace 95] Wallace, C.: "The semantics of the C++ programming language"; Börger, E. (ed.), Specification and Validation Methods, Oxford University Press (1995) 131-164.

[Winter 97] Winter, K.: "Model Checking for Abstract State Machines"; this volume

7 Appendix A. Summary of the GroundCELL Programs

[Feed Belt]

```
FB_NORMAL.
if currPhase = NormalRun and PieceInFeedBeltLightBarrier
then FeedBeltFree := True
     if TableReadyForLoading then currPhase := CriticalRun
                             else currPhase := Stopped

FB_STOPPED.
if currPhase = Stopped and TableReadyForLoading
then currPhase := CriticalRun

FB_CRITICAL.
if currPhase = CriticalRun and not PieceInFeedBeltLightBarrier
then currPhase := NormalRun
     TableLoaded := True

where TableReadyForLoading ≡ TableInLoadPosition and not TableLoaded
      TableInLoadPosition ≡ currPhase(ERT) = StoppedInLoadPosition
```

Initialization: currPhase = NormalRun , FeedBeltFree = true, PieceInFeedBeltLightBarrier = false

[Elevating Rotary Table]

```
WAITING_LOAD.
if currPhase = StoppedInLoadPosition and TableLoaded
then currPhase := MovingToUnloadPosition

MOVING_UNLOAD.
if currPhase = MovingToUnloadPosition and UnloadPositionReached
then currPhase := StoppedInUnloadPosition

WAITING_UNLOAD.
if currPhase = StoppedInUnloadPosition and not TableLoaded
then currPhase := MovingToLoadPosition

MOVING_LOAD.
if currPhase = MovingToLoadPosition and LoadPositionReached
then currPhase := StoppedInLoadPosition
```

Initialization: currPhase = StoppedInLoadPosition , TableLoaded = false

[Robot]

WAITING.
if currPhase = WaitingIn(UnloadTable|UnloadPress|LoadDepBelt|LoadPress)Pos
 ∧ (Table|Press|DepositBelt|Press)ReadyFor(Unloading|Unloading|Loading|Loading)
then currPhase := UnloadingTable |UnloadingPress |LoadingDepBelt |LoadingPress

ACTION.
if currPhase = UnloadingTable |UnloadingPress |LoadingDepBelt |LoadingPress
 ∧ (UnloadingTable |UnloadingPress |LoadingDepBelt |LoadingPress)Completed
then currPhase:=MovingTo(UnloadPress|LoadDepBelt|LoadPress|UnloadTable)Pos
  TableLoaded |PressLoaded |DepositBeltReadyForLoading |PressLoaded :=
   false|false|false|true

MOVING.
if currPhase = MovingTo(UnloadPress|LoadDepBelt|LoadPress|UnloadTable)Pos
 and (UnloadPress|LoadDepBelt|LoadPress|UnloadTable)PosReached
then currPhase:=WaitingIn(UnloadPress|LoadDepBelt|LoadPress|UnloadTable)Pos

where TableReadyForUnloading ≡ (TableInUnloadPosition and TableLoaded)
  TableInUnloadPosition ≡ (currPhase(ERT) = StoppedInUnloadPosition)
  PressReadyForUnloading ≡ (PressInUnloadPosition and PressLoaded)
  PressInUnloadPosition ≡ (currPhase(Press) = OpenForUnloading)
  PressReadyForLoading ≡ (PressInLoadPosition and not PressLoaded)
  PressInLoadPosition ≡ (currPhase(Press) = OpenForLoading)

Initialization: currPhase = WaitingInUnloadTablePos ,¬UnloadingTableCompleted

[Deposit Belt]

DB_NORMAL.
if currPhase= NormalRun and PieceInDepositBeltLightBarrier
then currPhase := CriticalRun

DB_CRITICAL.
if currPhase= CriticalRun and not PieceInDepositBeltLightBarrier
then currPhase := Stopped
  DepositBeltReadyForLoading := true
  PieceAtDepositBeltEnd := true

DB_STOPPED.
if currPhase= Stopped and not PieceAtDepositBeltEnd
then currPhase := NormalRun

Initialization: currPhase = NormalRun , PieceAtDepositBeltEnd = false,
DepositBeltReadyForLoading = true

[Press]

WAITING_UNLOAD.
if currPhase= OpenForUnloading and PressLoaded = false
then currPhase := MovingToMiddlePosition

MOVING_TO_MIDDLE.
if currPhase= MovingToMiddlePosition and MiddlePosition
then currPhase := OpenForLoading

WAITING_LOAD.
if currPhase= OpenForLoading and PressLoaded = true
then currPhase := MovingToTopPosition

MOVING_TO_UPPER.
if currPhase= MovingToTopPosition and TopPosition
then currPhase := ClosedForForging

CLOSED.
if currPhase= ClosedForForging and ForgingCompleted
then currPhase := MovingToBottomPosition

MOVING_TO_LOWER.
if currPhase= MovingToBottomPosition and BottomPosition
then currPhase := OpenForUnloading

Initialization: currPhase = OpenForUnloading , PressLoaded = true

[Traveling Crane]

WAITING(DB).
if currPhase= WaitingToUnloadDepositBelt and PieceAtDepositBeltEnd
then currPhase := UnloadingDepositBelt

UNLOADING(DB).
if currPhase= UnloadingDepositBelt and UnloadingDepositBeltCompleted
then currPhase := MovingToLoadFeedBeltPos
    PieceAtDepositBeltEnd := false

MOVING(FB).
if currPhase= MovingToLoadFeedBeltPos and LoadFeedBeltPosReached
then currPhase:= WaitingToLoadFeedBelt

WAITING(FB).
if currPhase= WaitingToLoadFeedBelt and FeedBeltFree
then currPhase := LoadingFeedBelt

LOADING(FB).
if currPhase = LoadingFeedBelt and LoadingFeedBeltCompleted
then currPhase := MovingToUnloadDepositBeltPos
    FeedBeltFree := false

MOVING(DB).
if currPhase= MovingToUnloadDepositBeltPos and UnloadDepositBeltPosReached
then currPhase := WaitingToUnloadDepositBelt

Initialization: currPhase = WaitingToUnloadDepositBelt ,
UnloadingDepositBeltCompleted = false

8 Appendix B. Summary of the RefinedCELL Programs

For reasons of space we refer to the main text for the declaration of functions (as monitored, controlled, interaction or derived).

[Feed Belt]

FB_NORMAL.
if NormalRun and PieceInFeedBeltLightBarrier
then FeedBeltFree := true
    if TableReadyForLoading then Delivering := true
    else FeedBeltMot := off

FB_STOPPED.
if Stopped and TableReadyForLoading
then FeedBeltMot := on
    Delivering := true

FB_CRITICAL.
if CriticalRun and not PieceInFeedBeltLightBarrier
then Delivering := false
    TableLoaded := True

where NormalRun ≡ FeedBeltMot = on and notDelivering
    CriticalRun ≡ FeedBeltMot = on and Delivering
    Stopped ≡ FeedBeltMot = off
    TableReadyForLoading ≡ TableInLoadPosition and not TableLoaded
    TableInLoadPosition ≡ StoppedInLoadPosition

Initialization: FeedBeltMot = on, Delivering = false,
FeedBeltFree = true, PieceInFeedBeltLightBarrier = false

[Elevating Rotary Table]

WAITING_LOAD.
if StoppedInLoadPosition and TableLoaded
thenTableElevationMot := Up
TableRotationMot := Clockwise

MOVING_UNLOAD.a.
if (TableElevationMot = Up) and TopPosition
then TableElevationMot := Idle

MOVING_UNLOAD.b.
if (TableRotationMot = Clockwise) and MaxRotation
then TableRotationMot := Idle

WAITING_UNLOAD.
if StoppedInUnloadPosition and not TableLoaded
thenTableElevationMot := Down
TableRotationMot := CounterClockwise

MOVING_LOAD.a.
if (TableElevationMot = Down) and BottomPosition
then TableElevationMot := Idle

MOVING_LOAD.b.
if (TableRotationMot = CounterClockwise) and MinRotation
then TableRotationMot := Idle

where StoppedInLoadPosition $\equiv$ (BottomPosition $\wedge$
MinRotation $\wedge$ (TableElevationMot = Idle) $\wedge$ (TableRotationMot = Idle))
StoppedInUnloadPosition $\equiv$ (TopPosition $\wedge$
MaxRotation $\wedge$ (TableElevationMot = Idle) $\wedge$ (TableRotationMot = Idle))

Initialization: TableElevationMot = idle, TableRotationMot = idle,
TableLoaded = false, MinRotation = true, BottomPosition = true

[Robot]

WAITING.
if WaitingIn(UnloadTable|UnloadPress|LoadDepBelt|LoadPress)Pos
$\wedge$ (Table|Press|DepositBelt|Press)ReadyFor(Unloading|Unloading|Loading|Loading)
then Arm(1|2|2|1)Mot := extend

ACTION.extension.
if Extending(Arm1ToTable |Arm2ToPress |Arm2ToDepBelt |Arm1ToPress)
$\wedge$ Arm(1|2|2|1)Ext =(OverTable |Arm2IntoPress |OverDepBelt |Arm1IntoPress)
then Arm(1|2|2|1)Mot := idle
Arm(1|2|2|1)Mag := on|on|off|off

```
ACTION.proper.
if Extended( OverTable |Arm2IntoPress |OverDepBelt |Arm1IntoPress )
  and Arm(1|2|2|1)Mag = on|on|off|off
then Arm(1|2|2|1)Mot := retract

ACTION.retraction.
if Retracting( Arm1ToTable |Arm2ToPress |Arm2ToDepBelt |Arm1ToPress )
  and Arm(1|2|2|1)Ext = retracted
then Arm(1|2|2|1)Mot := idle
     RobotRotationMot := counterClock|counterClock|counterClock|clockwise
     TableLoaded|PressLoaded|DepositBeltReadyForLoading|PressLoaded
       :=false|false|false|true

MOVING.
if MovingTo(UnloadPress|LoadDepBelt|LoadPress|UnloadTable)Pos
  and (UnloadPress|LoadDepBelt|LoadPress|UnloadTable)PosReached
then RobotRotationMot := idle

where WaitingIn(UnloadTable|UnloadPress|LoadDepBelt|LoadPress)Pos≡
          Angle = Arm1ToTable |Arm2ToPress |Arm2ToDepBelt |Arm1ToPress
           and ArmsRetracted and RobotIdle and Arm1Mag = off|on|on|on
           and Arm2Mag off|off|on|off
       Extending( Arm1ToTable |Arm2ToPress |Arm2ToDepBelt |Arm1ToPress )≡
          Angle = Arm1ToTable |Arm2ToPress |Arm2ToDepBelt |Arm1ToPress and
          Arm(1|2|2|1)Mot = extend
       Extended( OverTable |Arm2IntoPress |OverDepBelt |Arm1IntoPress )≡
          Angle = Arm1ToTable |Arm2ToPress |Arm2ToDepBelt |Arm1ToPress and
          Arm(1|2|2|1)Ext =(OverTable |Arm2IntoPress |OverDepBelt |Arm1IntoPress )
          and Arm(1|2|2|1)Mot = idle
       Retracting( Arm1ToTable |Arm2ToPress |Arm2ToDepBelt |Arm1ToPress )≡
          Angle = Arm1ToTable |Arm2ToPress |Arm2ToDepBelt |Arm1ToPress and
          Arm(1|2|2|1)Mot = retract
       (Arm1ToTable |Arm2ToPress |Arm2ToDepBelt |Arm1ToPress )Completed ≡
          Retracting( Arm1ToTable |Arm2ToPress |Arm2ToDepBelt |Arm1ToPress )
           and Arm(1|2|2|1)Ext = retracted
       MovingTo(UnloadPress|LoadDepBelt|LoadPress|UnloadTable)Pos ≡
          ArmsRetracted and Arm1Mot = idle and Arm2Mot = idle
          RobotRotationMot = counterClock|counterClock|counterClock|clockwise
          Arm1Mag = on|on|on|off and Arm2Mag = off|on|off|off
          Angle ∈ [Arm1ToTable , Arm2ToPress ]|
                  [Arm2ToPress , Arm2ToDepBelt ]|
                  [Arm2ToDepBelt , Arm1ToPress ]|
                  [Arm1ToTable , Arm1ToPress ]
       (UnloadPress|LoadDepBelt|LoadPress|UnloadTable)PosReached ≡
          Angle = Arm2ToPress |Arm2ToDepBelt |Arm1ToPress |Arm1ToTable
       ArmsRetracted ≡ Arm1Ext = retracted and Arm2Ext = retracted
       RobotIdle ≡ RobotRotationMot = idle ∧ Arm1Mot = idle ∧ Arm2Mot = idle
       TableReadyForUnloading ≡ (TableInUnloadPosition and TableLoaded )
       TableInUnloadPosition ≡ (TopPosition and MaxRotation )
```

PressReadyForUnloading ≡ (PressInUnloadPosition and PressLoaded)
PressInUnloadPosition ≡ (currPhase(P) = OpenForUnloading)
PressReadyForLoading ≡ (PressInLoadPosition and not PressLoaded)
PressInLoadPosition ≡ (currPhase(P) = OpenForLoading)

Initialization: Angle = Arm1ToTable , Arm1Ext = retracted,
Arm2Ext = retracted, RobotRotationMot = idle, Arm1Mot = idle,
Arm2Mot = idle, Arm1Mag = off, Arm2Mag = off

[Press]

WAITING_UNLOAD.
if OpenForUnloading and PressLoaded = false
then PressMot := up

MOVING_TO_MIDDLE.
if MovingToMiddlePosition and MiddlePosition
then PressMot := idle

WAITING_LOAD.
if OpenForLoading and PressLoaded = true
then PressMot := up

MOVING_TO_UPPER.
if MovingToTopPosition and TopPosition
then PressMot := idle

CLOSED.
if ClosedForForging and ForgingCompleted
then PressMot := down

MOVING_TO_LOWER.
if MovingToBottomPosition and BottomPosition
then PressMot := idle

where OpenForUnloading ≡ BottomPosition and PressMot = idle
MovingToMiddlePosition ≡ notPressLoaded and PressMot = up
OpenForLoading ≡ MiddlePosition and PressMot = idle
MovingToTopPosition ≡ PressLoaded and PressMot = up
ClosedForForging ≡ TopPosition and PressMot = idle
MovingToBottomPosition ≡ PressMot = down

Initialization: BottomPosition = true, PressMot = idle, PressLoaded = true

[Deposit Belt]

```
DB_NORMAL.
if NormalRun and PieceInDepositBeltLightBarrier
then Critical := true

DB_CRITICAL.
if CriticalRun and not PieceInDepositBeltLightBarrier
then DepBeltMot := off
     DepositBeltReadyForLoading := true
     PieceAtDepositBeltEnd := true

DB_STOPPED.
if Stopped and not PieceAtDepositBeltEnd
then DepBeltMot := on
     Critical := false

where NormalRun ≡
         DepBeltMot = on and not Critical
      CriticalRun ≡
         DepBeltMot = on and Critical
      Stopped ≡
         DepBeltMot = off
```

Initialization: DepBeltMot = on, Critical = false, PieceAtDepositBeltEnd = false, DepositBeltReadyForLoading = true

[Travelling Crane]

```
WAITING(DB).
if WaitingToUnloadDepositBelt and PieceAtDepositBeltEnd
then CraneMagnet := on

UNLOADING(DB).a.
if CraneVerticalMot = idle and GripperVerticalPos = OnDepositBelt and
  CraneMagnet = on
then CraneVerticalMot := up

UNLOADING(DB).b.
if CraneVerticalMot = up and GripperVerticalPos = SafeDistanceFromFeedBelt
then CraneVerticalMot := idle
     CraneHorizontalMot := toFeedBelt
     PieceAtDepositBeltEnd := false

MOVING(FB).
if CraneHorizontalMot = toFeedBelt and GripperOverFeedBelt
then CraneHorizontalMot := idle
```

WAITING(FB).
if WaitingToLoadFeedBelt and FeedBeltFree
then CraneVerticalMot := down

LOADING(FB).a.
if CraneVerticalMot = down $\wedge$ GripperVerticalPos = OnFeedBelt $\wedge$ GripperOverFeedBelt
then CraneVerticalMot := idle
    CraneMagnet := off

LOADING(FB).b.
if CraneVerticalMot = idle $\wedge$ GripperVerticalPos = OnFeedBelt $\wedge$ GripperOverFeedBelt
  and CraneHorizontalMot = idle and not CraneMagnet
then CraneHorizontalMot := toDepBelt
    FeedBeltFree := false

MOVING(DB).a.
if CraneHorizontalMot = toDepositBelt and GripperOverDepositBelt
then CraneHorizontalMot := idle
    CraneVerticalMot := down

MOVING(DB).b.
if GripperOverDepositBelt and
  CraneVerticalMot = down and GripperVerticalPos = OnDepositBelt
then CraneVerticalMot := idle

where WaitingToUnloadDepositBelt $\equiv$
    GripperOverDepositBelt and GripperVerticalPos = OnDepositBelt and
    CraneHorizontalMot = idle and CraneVerticalMot = idle and
    CraneMagnet = off
WaitingToLoadFeedBelt $\equiv$
  GripperOverFeedBelt and GripperVerticalPos = SafeDistanceFromFeedBelt and
  CraneHorizontalMot = idle and CraneVerticalMot = idle and
  CraneMagnet = on

Initialization: CraneHorizontalMot = idle, CraneVerticalMot = idle,
GripperOverDepositBelt = true, GripperVerticalPos = OnDepositBelt ,
CraneMagnet = off

9 Appendix C. Geometrical Layout of the Production Cell

The FZI simulator for the production cell can be accessed through

`http://www.fzi.de/divisions/prost/projects/production_cell`
`/ProductionCell.html`

The executable code for our program which controls this simulator is available at

`http://www.fzi.de/prost/projects/production_cell`
`/contributions/ASM.html`