MONITORING AND CHECKING OF DISCRETE EVENT SIMULATIONS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
THE MIDDLE EAST TECHNICAL UNIVERSITY

BY

BUKET ULU

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
IN
THE DEPARTMENT OF COMPUTER ENGINEERING

AUGUST 2003

Approval of the Graduate School of Natural And Applied Sciences.

Prof. Dr. Canan ÖZGEN
Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Master Science.

Prof. Dr. Ayşe KİPER
Head of Department

This is to certify that we have read this thesis and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master Science.

Assoc. Prof. Dr. Sinan KAYALIGİL Co-Supervisor	Assist. Prof. Dr. Halit OĞUZTÜZÜN Supervisor

Examining Committee Members:

Prof. Dr. Kemal LEBLEBİCİOĞLU

Prof. Dr. Faruk POLAT

Assoc. Prof. Dr. Sinan KAYALIGİL

Assist. Prof. Dr. Ahmet COŞAR

Assist. Prof. Dr. Halit OĞUZTÜZÜN

ABSTRACT

MONITORING AND CHECKING OF DISCRETE EVENT SIMULATIONS

Ulu, Buket
M.S., Department of Computer Engineering
Supervisor : Assist. Prof. Dr. Halit Oğuztüzün
Co-Supervisor : Assoc. Prof. Dr. Sinan Kayalıgil

August 2003, 127pages

Discrete event simulation is a widely used technique for decision support. The results of the simulation must be reliable for critical decision making problems. Therefore, much research has concentrated on the verification and validation of simulations. In this thesis, we apply a well-known dynamic verification technique, assertion checking method, as a validation technique. Our aim is to validate the particular runs of the simulation model, rather than the model itself.

As a case study, the operations of a manufacturing cell have been simulated. The cell, which is METUCIM Laboratory at the Mechanical Engineering Department of METU, has a robot and a conveyor to carry the materials, and two machines to manufacture the items, and a quality control to measure the correctness of the manufactured items.

This simulation is monitored and checked by using the Monitoring and Checking (MaC) tool, a prototype developed at the University of Pennsylvania. The separation of low-level implementation details (pertaining to the code) from the high-level requirement specifications (pertaining to the simuland) helps keep monitoring and checking the simulations at an abstract level.

Keywords : discrete event simulation, validation and verification, assertion checking, run-time monitoring, instrumentation.

ÖZ

KESİKLİ OLAYLARIN BENZETİMLERİ ÜZERİNDE İZLEME VE KONTROL

Ulu, Buket
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi : Y. Doç. Dr. Halit Oğuztüzün
Ortak Tez Yöneticisi : Doç. Dr. Sinan Kayalıgil

Ağustos 2003, 127sayfa

Kesikli olayların benzetimleri, yaygın olarak kullanılan bir karar destekleme tekniğidir. Kritik problemler için bu benzetimlerden elde edilen sonuçların güvenilirliği önem taşır. Bu nedenle, sistem benzetimlerinin doğrulanması ve geçerlenmesi üzerinde birçok araştırma yapılmaktadır. Bu tezde, çok iyi bilinen bir dinamik doğrulama tekniği olan, önesürümlerin denetimi yöntemi, benzetimlerin geçerliliğini kanıtlama yönünde uygulanmaktadır. Amacımız modelin kendisinden ziyade o modelin koşturmalarını geçerlemektir.

Bir vaka çalışması olarak, Ortadoğu Teknik Üniversitesi Makine Mühendisliği Bölümü Bilgisayar Tümleşik Üretim Laboratuvarı'nın bir benzetimi yapılmıştır. Bu laboratuvarda biri robot olmak üzere iki tane taşıyıcı, parçaları işleyen iki makine ve bir tane de kalite kontrol bileşeni bulunmaktadır.

Bu benzetimin izlenmesi ve kontrolü, Pensilvanya Üniversitesi'nde bu konuda geliştirilmiş bir ön ürünle yapılmaktadır. Bu çalışmada düşük-seviye olarak adlandırılan kod detayları, yüksek-seviye olarak adlandırılan sistem gereksinimlerinden ayrılmıştır. Bu da izlenme ve kontrolün soyut bir seviyede yapılmasını sağlamaktadır.

Anahtar Kelimeler : kesikli olayların benzetimleri, doğrulama ve geçerleme, önesürüm denetimi, koşturma sırasında izleme, enstrümantasyon.

To My Family

ACKNOWLEDGEMENT

I would like to acknowledge the help of many people during my study.

My supervisor, Halit Oğuztüzün, for bestowing his knowledge and time, providing resources, offering direction and insightful criticism, for numerous useful discussions, and also for his friendly manner;

My co-supervisor, Sinan Kayalıgil, for suggesting METUCIM Lab as a case study and all supports for this study;

Burak Sarı and Yusuf Başıbüyük, who are responsible for METUCIM Lab, for describing the laboratory, providing resources and data;

Central Bank of the Republic of Turkey and my superiors for supporting this academic study;

All my friends, for their valuable helps and smiling faces;

My brother, for always reminding me the future, and making my life colorful.

Finally, I am forever indebted to my parents for their understanding, endless love, encouragement and patience during my academic life.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

Verification and validation (V&V) have a big role in the simulation world. Verification means building the model right, and validation means building the right model. There are lots of V&V techniques; these are categorized as informal, static, dynamic and formal techniques.

Validation includes verification as a concept, because if a model is validated this means that it is already verified, but, on the contrary, if a model is verified, this does not mean that it is the right model. Therefore, validation gains more importance if the results of the simulations are used to support some decisions. Validation makes the results of a simulation model more reliable.

Assertion checking is one of the dynamic verification techniques. It is used as a verification method. In general, assertions are inserted into the code and in run-time, they are checked to see if they are satisfied.

In this thesis, we use the assertion checking method, which is one of the dynamic verification techniques, for validating the particular runs of a simulation model according to a set of requirement specifications. In dynamic techniques, simulation model behaviour is evaluated at run-time. For this purpose, we have employed a framework, Monitoring and Checking (MaC) framework, which has been developed at the University of Pennsylvania, and its prototype, Java-Mac release 0.99. This tool serves run-time monitoring of the target programs and checks the program according to defined requirement specifications.

Finally, to reach our aim, we had to simulate a system. We have chosen a real world system, METUCIM Lab in the Mechanical Engineering Department of METU, which is a single manufacturing cell supported by the computers. While analyzing the system, the validation requirements have been specified, and in the design phase, they have been changed from high-level to low-level. Then, this simulation has been checked against the specified validation requirements. For this purpose, different scenarios about METUCIM Lab have been created as if there are misconceptions about the real world. These scenarios have been run under the control of validation specifications, and from these runs, the results have been reported in this thesis.

The remainder of this thesis is organized as follows: In Chapter 2, background information is presented. Firstly, the concept of V&V, and their techniques are discussed. Then, the assertion checking method, run-time monitoring and instrumentation are described. Lastly, the architecture and the languages of MaC are covered.

In Chapter 3, the simulation model is introduced. For this purpose, firstly, the application domain, which is a manufacturing cell, is defined. Then, METUCIM Lab in the real world and the information about its model in the simulation world are described in detail.

In Chapter 4, the validation study about this simulation is written. The validation requirements are given group by group, and the scenarios, which were created to capture these requirements, and the results are described. The related source (PEDL and MEDL) files are presented in the appendices.

Chapter 5 is the conclusion and the discussion part of this thesis. In this chapter, the results of the validation studies are evaluated and discussed. Finally, some possible future work is suggested.

CHAPTER 2

BACKGROUND

2.1 *Verification and Validation*

Verification and validation are the most important concepts in the simulation world. They increase the reliability of the results of simulations and the critical decisions can be taken based on these results.

2.1.1 Basic Concepts

"Simulation models are increasingly being used in problem solving and in decision-making. The developers and users of these models, the decision-makers using information derived from the results of the models, and people affected by decisions based on such models are all rightly concerned with whether a model and its results are "correct." This concern is addressed through model verification and validation." [1]

"Model validation is substantiating that the model, within its domain of applicability, behaves with satisfactory accuracy consistent with the modeling and simulation (M&S) objectives. Model validation deals with building the *right* model." [2]

"Model verification is substantiating that the model is transformed from one form into another, as intended, with sufficient accuracy. Model verification deals with building the model *right*. The accuracy of transforming a problem formulation into a model specification or the accuracy of converting a model representation from micro flowchart form into an executable computer program is evaluated in model verification." [2]

Validation contains verification as a concept, because a validated system usually means that it is the right model, and built right. Therefore, such a system can also be expected that it is a verified model. However, a verified model cannot be said to be a right model, unless it captures all the specified requirements, which is so difficult to achieve. Also, from the meaning of verification, the priority is given to *building the model right*, verification deals with error-free implementation. In conclusion, if a validation condition (for example, the passengers must tie the safety belts before the airplane takes off) fails, it cannot be understood that the error is in the code or in the model. However, if the verification condition (for example, controlling the queue length while inserting a new element) fails, it is clear that there is something wrong with the code. [3]

2.1.2 Why is Validation Important?

The validation process increases the reliability of the model or the simulation of that model by reducing the risks. In some projects, simulations are used as a guide to the project development and management decision [6] . Therefore, in such projects, validation plays a critical role. Some examples for such projects are space searches and military works. Consequently, if a model is not validated, the results obtained from the simulation of this model are not reliable; but, if it is validated then it may be expected to behave like the real system modeled, and the decisions affecting the real system can be made with confidence based on the results of the simulation model [5] .

2.1.3 Special Challenges of Validation

Validating models or simulations has several challenges. These can be stated as below item by item [7]:

- Enough time and commitment must be obtained from the users to specify the validation requirements sufficiently and accurately. These requirements are stated often informally, incompletely and inconsistently.

- Different ideas about the represented requirements must be reconciled into a single complete and consistent set of requirements.

- Sufficient and accurate information describing the functionality and the performance of the modeled system must be obtained, especially if that system exists only conceptually.

- Validating a conceptual model described entirely informally and perhaps incompletely leaves a considerable number of parameters missing for qualitative interpretation about the degree to which the model faithfully represents the simulated system.

- Trying to balance project schedule and cost constraints against validation may result in a model or simulation with less validity than desired.

- Justifying the investment required to support the extent of the validation process is necessary to assure sufficient model or simulation credibility for the user.

- There is no general method for detecting incompleteness or inconsistency of requirement specifications.

2.1.4 V&V Techniques

Over seventy-five V&V techniques and eighteen statistical techniques that can be used for model validation are described in [6]. Most of these techniques are derived from software engineering; the remaining are specific to the M&S field. The V&V techniques can be separated into four categories: informal, static, dynamic, and formal.

Informal:

These techniques heavily rely on human reasoning and subjectivity. However, this does not mean that there is a lack of formal techniques. In fact, these techniques must be employed under well-structured formal guidelines. If they are used properly, their results can be very reliable.

Static:

These techniques do not require the execution of the programs, since in general, mental execution is applied. For these techniques, the accuracy of the model design and the source code becomes more important, because the aim is to check the information about the structure of the model, modeling techniques used, data and control flows within the model, and syntactical accuracy [4]. For example, the simulation language compiler is itself a kind of static V&V tool.

Dynamic:

Dynamic V&V techniques require the model execution and evaluate the model according to the execution behaviour. For evaluating the model while executing, most of the dynamic techniques need the model to be instrumented, which means insertion of additional codes (probes or stubs) into the executable code of the model. By this way, such techniques collect necessary information during the model execution.

Formal:

These techniques require that the correctness of the model must be proven by using the mathematical proof methods. The model development process must be well defined and structured for successful application of these techniques. Methods require the model development process to be well defined and structured. These techniques rather than the simpler methods can be applied to more complex problems. In fact, current formal proof correctness techniques cannot even be applied to a reasonable complex simulation; however, formal techniques can serve as the foundation for other V&V techniques.

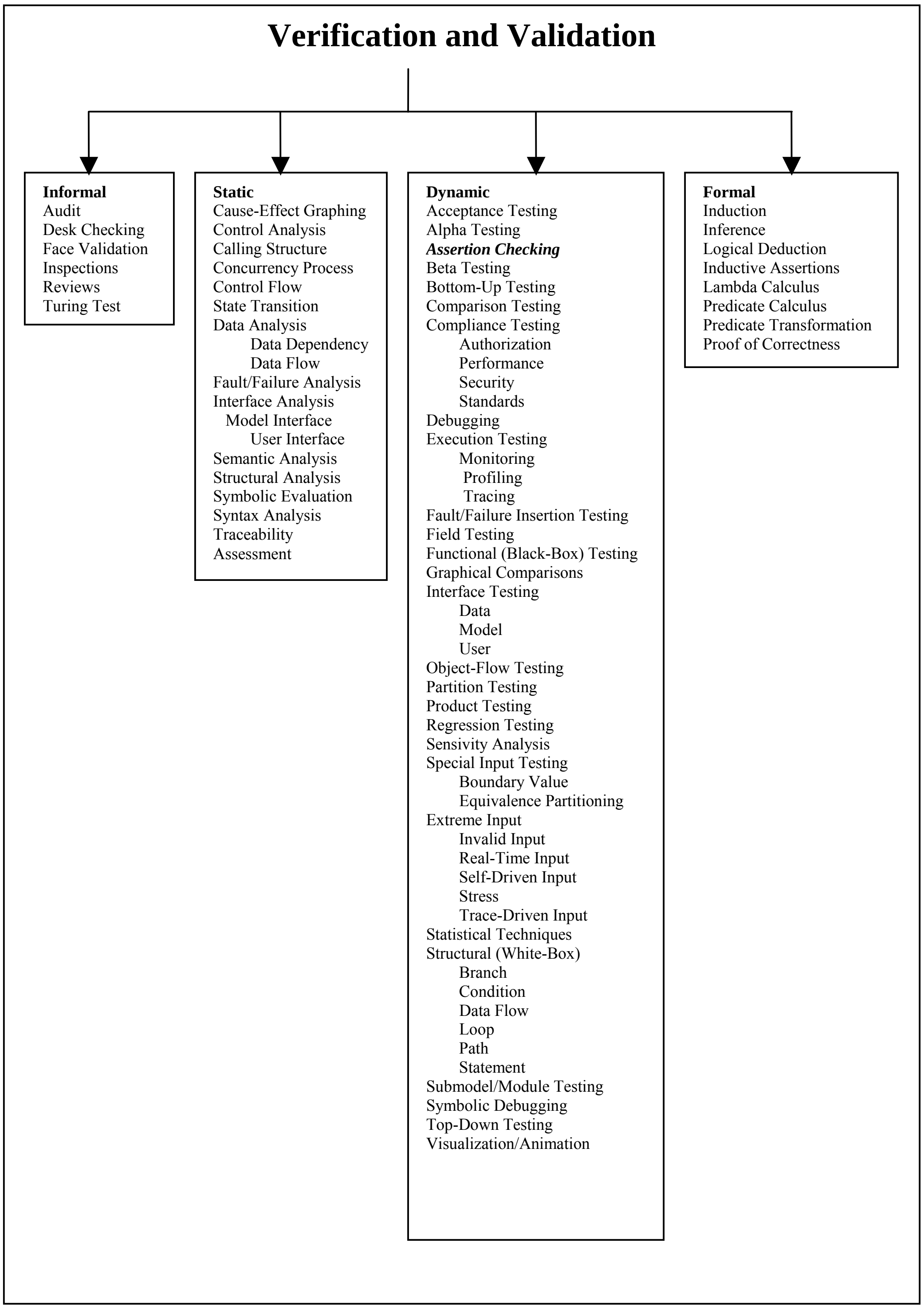

Figure 2.1 Taxonomy of Verification and Validation Techniques [6]

2.2 *Assertion Checking*

An assertion is a statement that must hold when a simulation model execution reaches that statement. It is a boolean expression decided to be true by the programmer. An assertion either evaluates to “true” when it is satisfied by the program state, to “false” otherwise [17].

Assertion checking is a dynamic verification technique in which assertions are placed in various parts of the software code to monitor its execution. The insertion of assertions into the user code, and the subsequent testing of such assertions at run-time is one of the most powerful techniques of verifying the software, although the additional codes make the runs slower. Even though, it is a verification technique, in this thesis, it is used for validation.

2.2.1 Assessment of Assertion Checking

There are some advantages and disadvantages of the assertion checking method; these are itemized below:

Advantages of Assertion Checking

1. It is a testing technique that will reveal defects early in the software development process.

2. It quickly uncovers the misconceptions of the programmer concerning the model being implemented.

3. Assertions can be monitored at run-time, making it a powerful debugging tool.

4. Adding assertions to the program may find faults where black-box test cases are ineffective.

5. Assertions can also be used to document and implement specifications, thereby facilitating the software development.

Disadvantages of Assertion Checking

1. There is a run-time cost associated with checking assertions. One way of getting a round this is by making it possible to selectively disable assertion checking. This can be achieved by using conditional compiler directives.

2. Leaving assertion checking in the code increases the size of object files. Despite this however, it is recommended that assertion checking remains enabled during development.

2.3 *Run-time Monitoring*

Run-time monitoring means evaluating code while it runs or scrutinizing the artifacts (event logs, etc) of running code. In this thesis, the first description is valid, the simulation runs are monitored, not the logs. It looks like debugging, but for this work, the meaning extracted from the values of the objects becomes valuable. For example, suppose that in a program execution, the value of the variable x is increased by 1 and the initial value of x is 0. If x is 0 or 1, this has the meaning of "x *is less than 2*". If x is 2, this means that "x *is equal to 2*". If x is 3 or 4, the meaning is that "x *is greater than 2*". Therefore, the aim of the monitoring is to reach meaningful information, which is the validation assertions of the software, for this work.

Benefits

- Requires a relatively small incremental effort over traditional testing.
- Combines the ease of testing with the power of formal methods.
- Can locate error potential for problems that test engineers may not envision.

Challenges

- Logic-based monitoring can add overhead to the normal execution of programs.
- While detecting difficult to find errors, error pattern runtime analysis can also detect problems that do not exist (false positives).

The inherent limitation of the run-time monitoring is that it observes the current program execution, but cannot observe all possible runs. Therefore, we cannot directly address the issue of model validation. As indicated above, only the particular runs are checked because of this limitation.

2.4 *Instrumentation*

Program instrumentation means inserting additional code to an existing program and in such a way that collecting necessary information is possible while the program runs. These additional codes are inserted at the points of the statements, which are critical for monitoring. Finally, these instrumented code runs and monitoring is done by the help of those inserted codes.

“An important characteristic is the *abstraction level* at which the program instrumentation is done. This abstraction level can be the hardware level, the library level, the source code level or the machine instruction level. It turns out that the only true viable instrumentation method is dynamic instrumentation: instrumentation code is added to the executable while it is running [9]. This way, also the dynamic linked libraries can be instrumented”. [8]

2.5 *Monitoring and Checking (MaC) Framework*

2.5.1 MaC Architecture

The Monitoring and Checking (MaC) architecture has been developed at the University of Pennsylvania, with the aim of guaranteeing the correctness of a run of the target program according to a formal requirement specification [10].

This is a general architecture, independent from any programming language. However, to demonstrate the effectiveness of this architecture, a prototype has been implemented for Java programs, which is called Java-MaC. This prototype has a feature of automatic instrumentation of Java byte-codes. Also, other run-time components to monitor and check the run of an instrumented program are generated automatically for easy deployment of Java-MaC.

The structure of the MaC architecture is illustrated in Figure 2.2. The architecture has two main phases: static and run-time phase. In the static phase, the run-time components, namely a filter, an event recognizer and a run-time checker, are automatically generated from a target program and formal requirement specifications. In the run-time phase, necessary information is collected from the running program and checked against the given formal requirement specification [11].

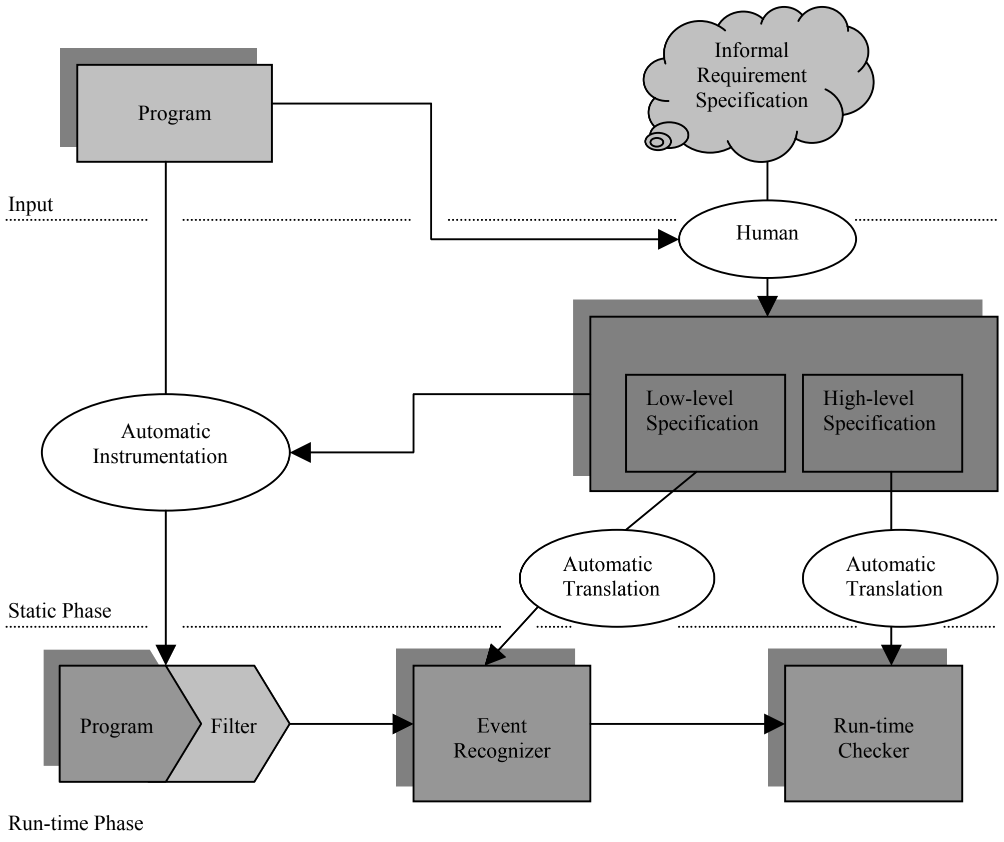

Figure 2.2 Overview of the MaC Architecture [11]

2.5.1.1 Static Phase

In this phase, there is a mapping between high-level events used in the high-level requirement specification and low-level state information extracted from the instrumented target program during execution. These are related explicitly by means of a low-level specification, which describes how events at the high-level requirement are defined in terms of monitored states of a target program. For example, in a simple manufacturing cell, the requirements may be expressed in terms of the condition system_capacity_exceeded. The target program, on the other hand, stores the count of items in the system in an item_count. The low-level specification in this case defines the event system_capacity_exceeded as item_count > 14.

To achieve the monitoring and checking, two components are created in this phase. One of them is a filter and used for monitoring. It is generated from the low-level specification. The second one is the event checker and used for checking the specified requirements. It is generated from the high-level specification. Both of these are created automatically. How they work is described in the run-time phase below.

2.5.1.2 Run-time Phase

During the run-time phase, the instrumented code is executed and meanwhile it is monitored and checked with respect to the requirement specification. The filter sends relevant state information to the event recognizer and by using this information event recognizer determines the occurrence of events. These events are then relayed to the run-time checker to check whether there is a violation of requirements or not.

Filter

The target program is instrumented directly by inserting additional codes on its executable code (in Java, byte-code) according to the low-level description in the monitoring script. The set of such program fragments inserted into the target code are called a filter. A filter keeps track of changes of monitored objects and sends related state information to the event recognizer.

Event Recognizer

The event recognizer receives the values of monitored objects from the filter and according to these values, it detects that there is an event occurred or not. It changes low-level information into high-level information. Events, which wanted to be recognized, are written into the monitoring script by using

Primitive Event Definition Language (PEDL) and these are relayed to the run-time checker. The event recognizer is the part of the monitoring and also it can be combined with the filter. The reason behind thinking them separately is to provide flexibility in an implementation of the architecture.

Run-time Checker

The run-time checker receives the necessary information from the event recognizer and by evaluating this, it decides whether or not the current execution history satisfies the given requirements written by using Meta Event Definition Language (MEDL). Also, the run-time checker does not only give warning; if there is a violation which is not dangerous, it may perform some recovery operations; which are written in a script by using Steering Action Definition Language (SADL).

2.5.2 MaC Languages

Before giving the description of the languages, we should discuss about events and conditions. Events are either present or not at an instant of time, they depend on the state changes. On the other hand, the conditions are true or false for a finite time duration. They also depend on the state changes, but between two state changes they are present. For example, machine_full_event event occurs whenever the monitored machine_full variable value becomes true, (machine_full==true); machine_full_condition holds true, while this variable value remains true, or until it leaves the state of being true. For the last sentence starting with "until", it can also be said that "until it becomes false"; however, this is not correct, because while the program reaches such a state that this variable is out of that scope, then it becomes undefined. Therefore, in Java-MaC this is handled by evaluating the logical expressions over 3 values: true, false and undefined (Λ).[12]

If we assume a countable set $C = \{c_1, c_2, \ldots\}$ of primitive conditions and a countable set of $\epsilon = \{e_1, e_2, \ldots\}$ of primitive events, the syntax of conditions and events is specified as below:

<*C*> ::= *c* | defined(<*C*>) | [<*E*>,<*E*>) | !<*C*> | <*C*> && <*C*> | <*C*> || <*C*> | <*C*> => <*C*>
<*E*> ::= *e* | start(<*C*>) | end(<*C*>) | <*E*> && <*E*> | <*E*> || <*E*> | <*E*> when <*C*>

- As indicated above, because the condition can have an undefined value, to handle this, **defined(*c*)** is used, that is true whenever the condition *c* has a well-defined value (true or false).

- By using two events, an interval of time can be defined, so a condition **[*e₁*,*e₂*)** is true from the event e_1 until the event e_2.

- **start(*c*)** is an event definition, which depends on an instant of time when the condition *c* starts to be true.

- **end(*c*)** is an event definition, which means that at that instant of time the condition *c* starts to be false. Also, an event, which depends on the instant of the condition whose value is Λ, is defined as **end(defined(*c*))**.

- **(*e* when *c*)** is an event that present if the event ***e*** occurs when the condition ***c*** is true.

Also, Java-MaC has **time** attribute for events. **time(*e*)** is the time when the event ***e*** occurs, according to the clock of the monitored system(this can be different from the clock of the monitor).

The reason why the languages in the MaC framework are called “event definition languages” is that when any condition becomes true, false or Λ, this can be identified as an event.

2.5.2.1 Primitive Event Definition Language (PEDL)

PEDL is the language for writing the monitoring scripts, which have low-level requirement specifications, so the implementation-specific details of the target program are used. The attributes and the methods, which need to be monitored, are written and, by using these, the events and conditions are defined. The name of this language indicates that the main purpose of PEDL specifications is to define primitive events of the requirement specifications. From these primitive events, more complex events can be defined. The design of PEDL has two principles:

1. Encapsulating all implementation-specific details of the monitoring process in PEDL specifications.

2. Extract only the current state of the target program execution to make the process of event recognizer as simple as possible.

2.5.2.2 Meta Event Definition Language (MEDL)

With this script, which is written in MEDL, the correctness of the execution system is checked by using the safety requirements, which must hold true, and the alarms, which must not raise during the execution. MEDL is also based on events and conditions. Primitive events and conditions defined in PEDL are imported, so the language has the name of ”meta event definition”. By using these imported events and conditions new events and conditions are defined. Also, in this script, auxiliary variables can be defined, and their values can be updated, and this is bound to the occurrence of events. In the new conditions and the events, these variables can be used, also. The high-level requirements of the system are defined by using safety properties and alarms.

2.5.2.3 Steering Action Definition Language (SADL)

When steering is wanted to be used, a steering script is written in SADL, besides the monitoring script. Steering actions and conditions are described in this script. If requirement violations detected by the event checker occur, then steering actions are invoked in response to these violations. Additional instrumentation date is generated by the steering script and also a special run-time component is created by this script which called *injector* that accepts action invocations from the monitor and triggers their execution within the system.

2.5.3 Example

In this section, an example (inspired by our application) to illustrate the use of PEDL and MEDL is given. This example is about a robot, a machine and lots of parts in a buffer some of which must be processed by this machine, some of which must not. If the current part will not be manufactured by the machine, robot passes to the next part until finds such a part for the machine. After finding, it puts the part into the machine, machine works, and after machine finishes to manufacture the part, robot takes the part and puts another buffer, and then this process starts from the beginning. The requirement of the system is that while the machine is full, the robot cannot try to put a new part into it. It is assumed that, there are two classes in the system, Robot and Machine. Both has an attribute full, which means that if full is true, then it is full. The robot has request_type attribute, because all the parts do not request the same thing, only some them request to be manufactured in the machine.

```
class Robot{
            boolean full;
            int request_type;
            ….
```

```
        public void main(String[] args){
        …..
        }
}

class Machine{
        boolean full;
        ….
}
```

The below is PEDL script. Three attributes and one method, which is the main method, are monitored. By using the values of the attributes, three conditions are defined and two of them are exported to the MEDL script. The start of the main method is also monitored to recognize the beginning of the execution. The following PEDL script introduces three high-level events, and three high level conditions.

```
MonScr Example
  export condition machineFull_cond, robotEmpty_cond;
  export event startPgm, start_put_machine, machineEmpty_event;

  //Monitored Variable Declaration:
  monobj boolean Machine.full;
  monobj boolean Robot.full;
  monobj int Robot.request_type;

  //Monitored Method Declaration:
  monmeth void Robot.main(String[]);

  //Condition Definitions:
  condition machineFull_cond = (Machine.full == true);
  condition robotEmpty_cond = (Robot.full == false);
  conditon req_put_machine= (Robot.request_type ==2);

  //Event Definitions:
  event startPgm = start(Robot.main(String[]));
```

```
    event start_put_machine = start(req_put_machine) &&
                                          end(robotEmpty_cond);
   event machineEmpty_event = end(machineFull_cond);
End
```

The requirement specification for this example is indicated above, a part can be put into the machine while it is empty. Also, how many parts are processed by the machine can be calculated by using an auxiliary variable to keep the this count. When the execution starts it is given an initial value, 0, and every time the machine becomes full, it is incremented by 1. The condition putting is defined as from robot starts to put a part until it becomes empty (means that it has put the part). The MEDL script is below:

```
ReqSpec Example
   import condition machineFull_cond, robotEmpty_cond;
   import event startPgm, start_put_machine, machineEmpty_event;

   //Auxiliary Variables:
   var int machine_part_count;

   //Condition Definitions:
   condition putting = [start_put_machine, robotEmpty);

   //Event Definitions:
   event robotEmpty = start(robotEmpty_cond);
   event machineFull_event = start(machineFull_cond);

   //Safety Property Definitions:
   property safeExample = putting ->  !machineFull_cond ;

   //Guards:
   startPgm -> { machine_part_count=0; }
   machineFull_event -> { machine_part_count=machine_part_count +1; }
End
```

CHAPTER 3

SIMULATION MODEL

In this thesis, the operation of METUCIM Lab in the Mechanical Engineering Department of METU is simulated. METUCIM Lab is a single manufacturing cell. Therefore, in this chapter, the basic definitions about manufacturing cell are given, firstly. Afterwards, its real world model, and the simulation model are described in detail.

3.1 *Application Domain: Manufacturing Cell*

3.1.1 Manufacturing Cells

A manufacturing cell is a series of automatic machine tools or items of fabrication equipment linked together with an automatic material handling system. Their shop floor layout is usually classified as below [16]:

Function: According to the functionality, similar machines are grouped together, to perform one specific machining or inspection task. These functional groups are then located relative to each other to minimize interdepartmental material handling.

Line: If there is a strict order or sequence required, a line or flow arrangement of machines can be applied, which is also called as transfer line. In this layout, machines are located next to each other in the order of their usage.

Cell: The cell layout is a combination of both the function and line layouts to use the efficiencies of both by decreasing into a single multifunctional unit. Sometimes it is called as a Group Technology Cell, each individual cell or department has different machines such that no machines may similar to any in located in the other cell or department.

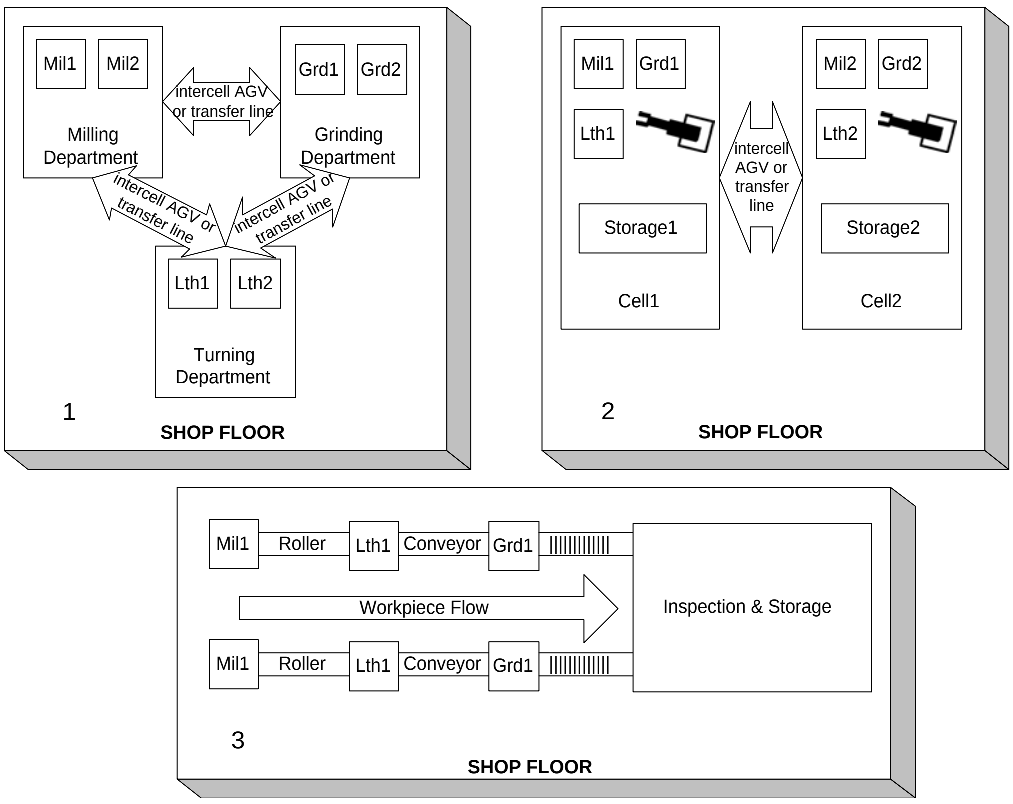

Figure 3.1 Shop floor arranged in functional (1), cellular (2), line (3) layouts [13]

Each layout brings certain advantages and disadvantages, and the optimum solution highly depends on the application. In any arrangement, a flexible manufacturing cell (FMC) is the latest application of computer control to automate manufacturing to achieve higher productivity and flexibility from the manufacturing equipment [15].

3.1.2 Computer Integrated Manufacturing (CIM)

Computer Integrated Manufacturing is total or near total integration of all computer systems in a manufacturing facility for free exchange of information and sharing of databases. This integration may extend beyond the confines of one factory into multiple manufacturing facilities and into the facilities of customers and vendors. CIM is an approach to the organization and management of a firm, in which all functions including order processing, design, manufacturing and production management, accounting, finance and computerized equipment on the shop floor are completely coordinated, through the use of computers and information/communication technologies. The idea is to form one large computer integrated system that connects all activities, and by doing this, common information is shared on a real-time basis for decision-making and control [16].

3.3 *Process-Oriented Model of METUCIM Lab*

3.3.1 METUCIM Lab

METUCIM basically consists of a single manufacturing cell and is classified as a “Flexible Manufacturing Cell”.

The main material handling system is the closed loop buffer, which is called conveyor and the 6-axis robot. Also, there is a static input/output buffer for loading and unloading parts to the system. The robot can move between the Computerized Numerical Control (CNC) Turning and CNC Milling Machine over the Pneumatic Linear Robot Drive (PLRD). Coordinate Measuring

Machine (CMM) controls the quality of the manufactured items in the cell. The main job of the system is processing the small parts to give them a wanted shape. For example, chessmen can be made from brass. A general view of the system is in Figure 3.2.

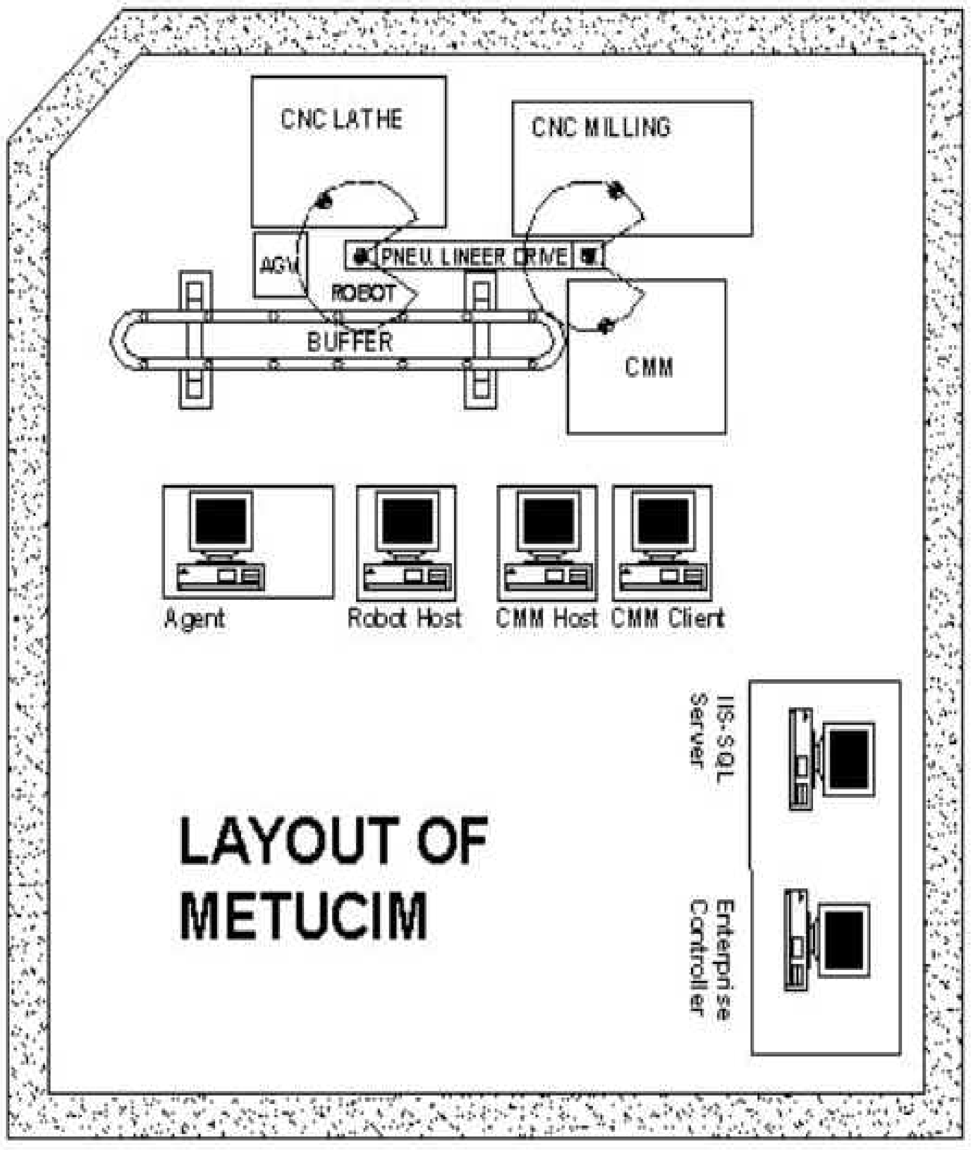

Figure 3.2 A general view of METUCIM Lab [14]

Functionality, properties and capabilities of the manufacturing, transport and quality control hardware can be summarized as [14]:

1. **CNC Turning Machine:** Mirac/Denford/UK. PC based, medium duty lathe having 2 simultaneously controlled axes. Equipped with a turret having 8 stations. Door and chuck are pneumatically powered. Can handle typically bars up to 50 mm in diameter and 150 mm in length, speeds up to 2500 rpm. Has a user-friendly built-in interface to visualize and debug part programs. The control is via standard RS 232 serial communication port and I/O card at a single sensor channel. Channel state OFF indicates that there is no part program running, or the task is finished. Channel state is ON when there is an active program running. “M62” and “M64” codes make the channel ON and OFF respectively.

2. **CNC Milling Machine:** Triac/Denford/UK. PC based, medium duty milling machine having 3 simultaneously controlled axes. Equipped with an automatic tool magazine with 6 stations. Door, chuck and tool magazine are pneumatically powered. Can handle parts up to 200 mm in width and 500 mm in length, speeds up to 2500 rpm. Has a user-friendly built-in interface to visualize and debug part programs. The control is via standard RS 232 serial communication port and I/O card at a single sensor channel. Channel state OFF indicates that there is no part program running, or the task is finished. Channel state is ON when there is an active program running. “M62” and “M64” codes make the channel ON and OFF respectively.

3. **Coordinate Measuring Machine (CMM):** Kemco/UK. 3 Axis CMM with a table of 600x400 mm wide. Capable of measuring up to 1μm resolution and accuracy. The control is accomplished using its own feature based control software running at its host computer. Coupled to the client via the

standard RS 232 serial communication port. The client is the part of the agent-based system through Ethernet communication.

4. **Closed Loop Buffer:** SKF/UK. Unidirectional, constant speed, closed loop buffer having 14 cups. Typically, it can handle cylindrical parts up to 50 mm in diameter. Makes a full rotation in 1.5 minutes approximately. Driven by a motor with gearbox. The control is via 48 channel I/O card. Has one operate channel and one counter channel. When the operate channel is ON, it starts to rotate and stops when the channel is OFF, the counter channel is used to count the cups passed.

5. **Robot:** Movemaster EX/Mitsubishi/Japan. 6 axis controlled material handling robot. Capable of handling bars of 50 mm in diameter, weight of 3 kg approximately. The control is by storing positions taught by the user in its EPROM and they can be executed by external triggering of program commands through RS232 connection from the computer. A DSR (data set ready) signal from the serial port indicates that there is no active program running or the task is finished.

6. **Pneumatic Linear Robot Drive (PLRD):** FESTO/Germany. Pneumatically powered linear drive for the robot. Has a movement range of 2m. Has two stop positions at both ends only. In METUCIM configuration it is used to move the robot from CNC Turning to CNC Milling/CMM neighborhood. The control is via 48 channel I/O card. Has two operate- and two sensor channels. When the first operate channel is triggered and immediately released it moves to right and vice versa for the second. Sensor channels on the left- and right positions indicate ON when the robot is at left and right ends of its range respectively.

7. **Static Buffer (AGV):** Buffer used for in and out loading to the cell. Has 3 input stations, which can handle bars of 70-90-100 mm, and 3 output buffers, Accept, Reject and Rework respectively. Is not physically connected or driven by a computer, but as the agent, status information is kept. Although it has no computer control and moving capabilities, it is modeled as an AGV in the system.

METUCIM Laboratory is a computer supported manufacturing cell. In fact, computers are the essential parts of METUCIM. Also, since it is a flexible manufacturing cell, the hardware architecture should support long-term flexibility also for future modifications. Therefore, while describing how this cell works, the software part is ignored; only the work viewed by an outside observer is described.

3.3.2 How does METUCIM work?

There is an input-output buffer (AGV) in the cell. The parts enter into the cell from input buffer and exit the cell from output buffer. There is no outside intervention of the operation of the cell; the cell interacts with the outside world only through input-output buffer (iobuffer). The carriers are robot and the closed loop buffer, conveyor. There are 14 cups on the conveyor, and each cup can hold only one part at the same time. The conveyor has three fix positions for the robot: iobuffer matching position, CNC1 matching position and CNC2 matching position. While loading/unloading processes, the conveyor must fix its cups to those specific positions. When the robot loads one of the three cups, they must be free. Also, when the robot unloads these cups, they must be full. The other details are told later under this subtitle.

How does METUCIM manufacture a part, which is processed by CNC1 and CNC2? Here is the dynamic view of METUCIM Lab: When the cell starts to work, there must be no part on it; all machines and carriers must be idle. The part, which will be manufactured, must be in the input buffer. The robot takes the part from the input buffer, and puts it onto the cup, which is at the iobuffer entry position. After loading this part, the robot gets its waiting position. The robot does this after every load/unload operation. The conveyor rotates by one cup for indexing the part to the CNC1 entry position. The robot loads/unloads CNC1 only from this entry. The capacity of the CNC1 is limited by one part. Therefore, when robot tries to load CNC1, it must be free. After the robot puts the part into it, it gets its waiting position and CNC1 starts to process the part. When CNC1 finishes its job, the robot takes the part from it and puts onto the cup at the CNC1 entry position of the conveyor. Then, the conveyor rotates by 3 cups to fix the part to the CNC2 entry position, because CNC2 and CMM are loaded/unloaded by using this entry position. The robot takes its position on the right side of the cell and puts the part into the CNC2. After CNC2 finishes processing the part, the robot takes it from CNC2 and puts onto the cup at the CNC2 entry position again. Then, to put the part into the CMM machine, it takes the part from this position. Also, CNC2 and CMM must be free, when the robot tries to load them; because their capacities are limited by one part, too. After measured by CMM, the part is taken and reloaded on the cup at CNC2 entry position. Then, the conveyor rotates by 10 cups to fix the part to the iobuffer entry position. The robot comes on the left side of the cell and puts the part onto the output buffer. This means that the part exits the cell.

The important points from the above paragraph are:

- First of all, the cell does not perform any useless operation: Any action of a component contributes the fulfillment of some request.

- Robot loads/unloads the buffers and the machines always from the conveyor. It is not possible to directly carry a part from one machine or buffer to the other.

- The conveyor has three machine matching positions. The robot uses iobuffer entry position for unloading input buffer and loading output buffer. It uses CNC1 entry position for loading/unloading CNC1, and CNC2 entry position for loading/unloading CNC2 and CMM.

- The capacity of the conveyor is 14, however for CNC1, CNC2 and CMM, it is one. These machines can process only one part at the same time. Also, the robot can carry only one part. The input and output buffers are assumed to have infinite capacities.

- In a carrying operation of the robot, the source must be full, and the target must be empty.

3.3.3 Simulation Model

The simulation model of METUCIM Lab has been built according to the outside observation as discussed above with some assumptions. The actual cell performs sequential processing, i.e. after one part is manufactured in the cell, the other enters. In the simulated cell, multiple parts can be processed in parallel. However, it capacity is limited; at the same time, there can be at most 14 parts in the cell; it does not accept the fifteenth one until any part leaves the cell.

The robot is the moving component of the cell, so the requests coming from the other components are evaluated only by the robot. The requests can be one of the below ones:

a. **IOBUFFER :** take part from IOBUFFER and load it onto conveyor
b. **CNC1 :** take part from CNC1 and load it onto conveyor
c. **CONVEYOR_IOBUFFER :** take part from conveyor and load it onto output buffer

d. **CONVEYOR_CNC1 :** take part from conveyor and load it into CNC1
e. **CONVEYOR_CNC2 :** take part from conveyor and load it into CNC2
f. **CONVEYOR_CMM :** take part from conveyor and load it into CMM
g. **CNC2 :** take part from CNC2 and load onto conveyor
h. **CMM :** take part from CMM and load onto conveyor

Three part types are defined since there are two CNC machines in this cell:

a. **type 1 :** This part type is processed first by CNC1 and then CNC2.
b. **type 2 :** This part type is processed by only CNC1.
c. **type 3 :** This part type is processed by only CNC2.

All parts are measured by CMM before leaving the system and all of them leave the system whether they are manufactured correctly or not. Although there are three output buffers according to the result of the CMM, Accept, Reject and Rework buffers, it is assumed that they are combined into one. Parts are processed according to their types. Parts of the same type are considered identical. The system processed the first nearest part on the conveyor. Therefore, there is a possibility that the first entering part of the run can be rotate over the conveyor by many without processed until the run ends and it can be the last part leaving the system. Also, for the runs with multiple parts, the movement of the conveyor is such that if the robot wants to put a part onto the conveyor and that specific cup is full, conveyor rotates to index the first free cup to that entry. If the requested part type is not on that specific cup, then the conveyor rotates to fix the nearest cup with a part of requested type to that index. Finally, the cell starts to run with no parts, and it runs for 7 hours without any failure (particularly, no component breaks down), and after that duration, it stops accepting new parts; it only tries to manufacture the remaining parts. After all parts exit, then it stops in an idle state.

3.3.3.1Use Case, Class and Sequence Diagrams

This model was implemented by using Java, because of the MaC's tool. In this simulation model, multithreading system has been used. There are nine classes four of which, namely Customer, IOBuffer, Robot and Cnn_Cmm classes, extend the *thread* class. The classes of the model are described in detailed below:

1. **Request :** It has been created to define the attributes of a request.

2. **RequesterQueue :** All requests made while the system runs is kept in this queue as a 'first in first out' manner. Only the robot is allowed to perform a dequeue operation on this queue.

3. **METUCIM :** This class starts a simulation run.

4. **Customer :** This class puts new parts onto the input buffer at certain time instants, and signals this buffer of the system.

5. **IOBuffer :** The raw parts and the manufactured parts are put into this buffer. It uses the special entry of the conveyor: IOBuffer entry.

6. **Robot :** This class is the core of the system. Robot evaluates the requests. If it cannot fulfill this request then it reinserts the request into the queue, else it signals the machine in which the part will be processed. For example, if the current request is CONVEYOR_CNC1, but CNC1 is full, then the robot reinserts this request into the queue. Each time this request is evaluated by the robot, it is reinserted until CNC1 becomes empty.

7. **Conveyor :** According to the requests evaluated by the robot, it rotates by certain cups to fix a free cup or the first part, which matches the request.

8. **Cnc_Cmm :** The behavior of CNC1, CNC2 and CMM is all same. Therefore, this is the general class of the machines processing the parts. To represent these three machines, this class used as a parent class and three new classes inherit this one. However, they can be ignored while counting the number of classes for the model. The main working concept behind these classes is that they all wait for the signal, which will be coming from the robot. There are two specific conveyor entries for transferring parts between these and the conveyor. CNC2 and CMM machines use the same entry.

9. **Signal :** Because there are thread classes, this class controls and arranges the all system signals.

The use case diagram of the model is in Figure 3.3 and the class diagram is in Figure 3.4. In the class diagram, CNC1, CNC2 and CMM are seen as different classes, which inherit the Cnc_Cmm class. In fact, because of the MaC's limitations, the Cnc_Cmm class has been erased, and three exact copies of classes whose names are Cnc1, Cnc2 and Cmm have been created. However, the diagrams for the simulation are created before starting to use MaC.

The sequence diagrams of each part type are in figures 3.5 through 3.7.

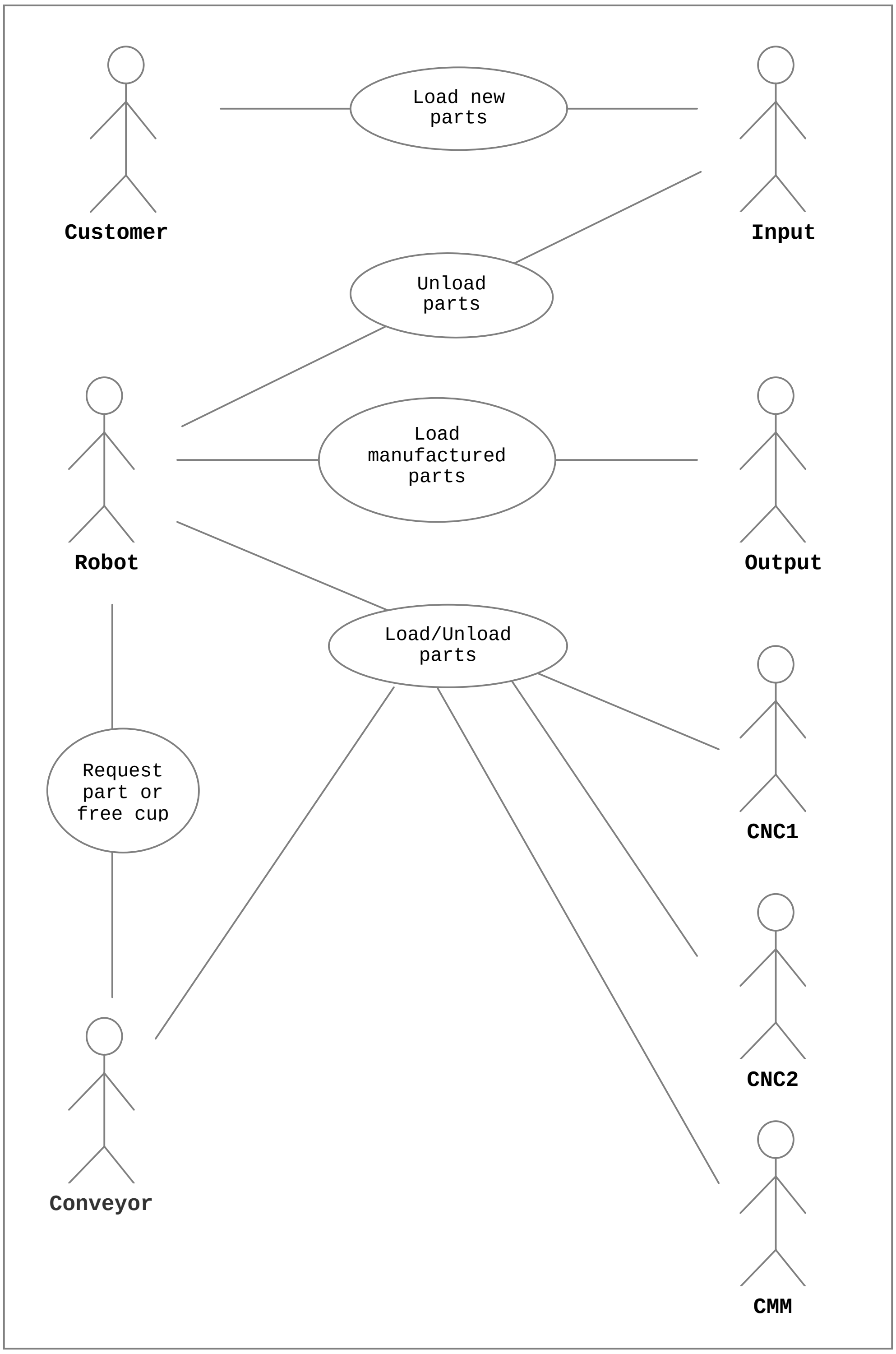

Figure 3.3 Use Case Diagram for METUCIM Lab Simulation Model

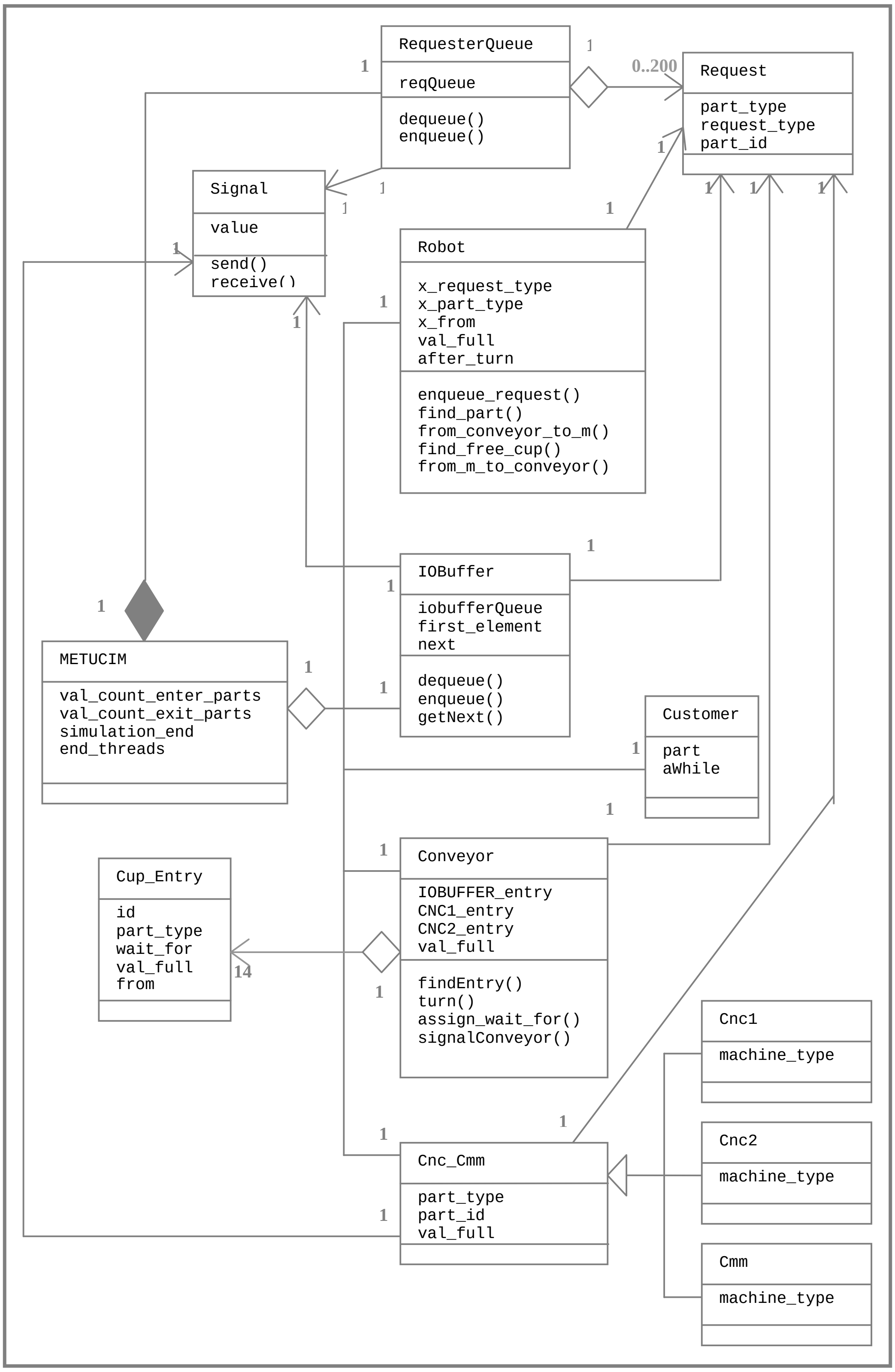

Figure 3.4 Class Diagram for METUCIM Lab Simulation Model

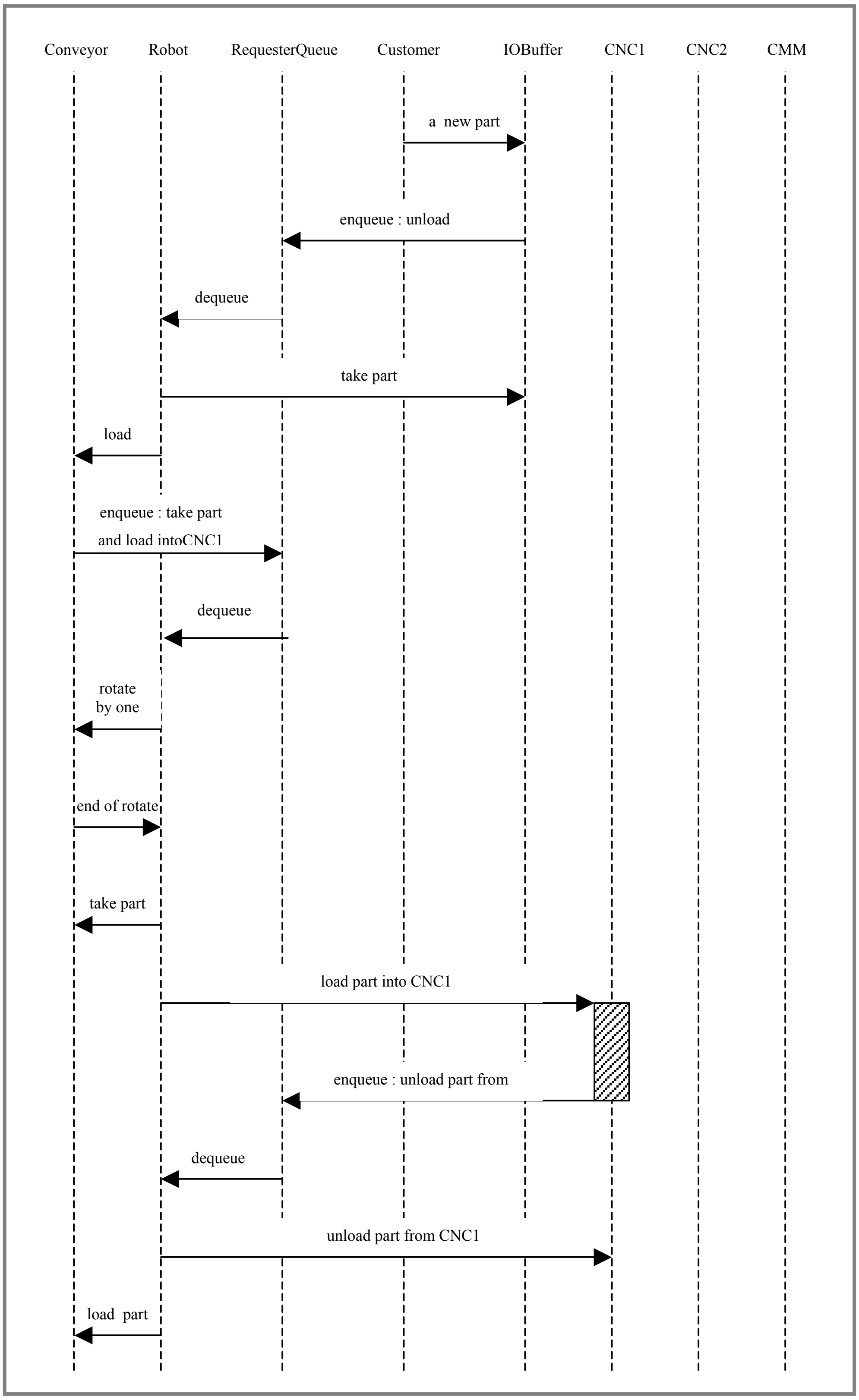

Figure 3.5 Sequence Diagram of Part Type 1

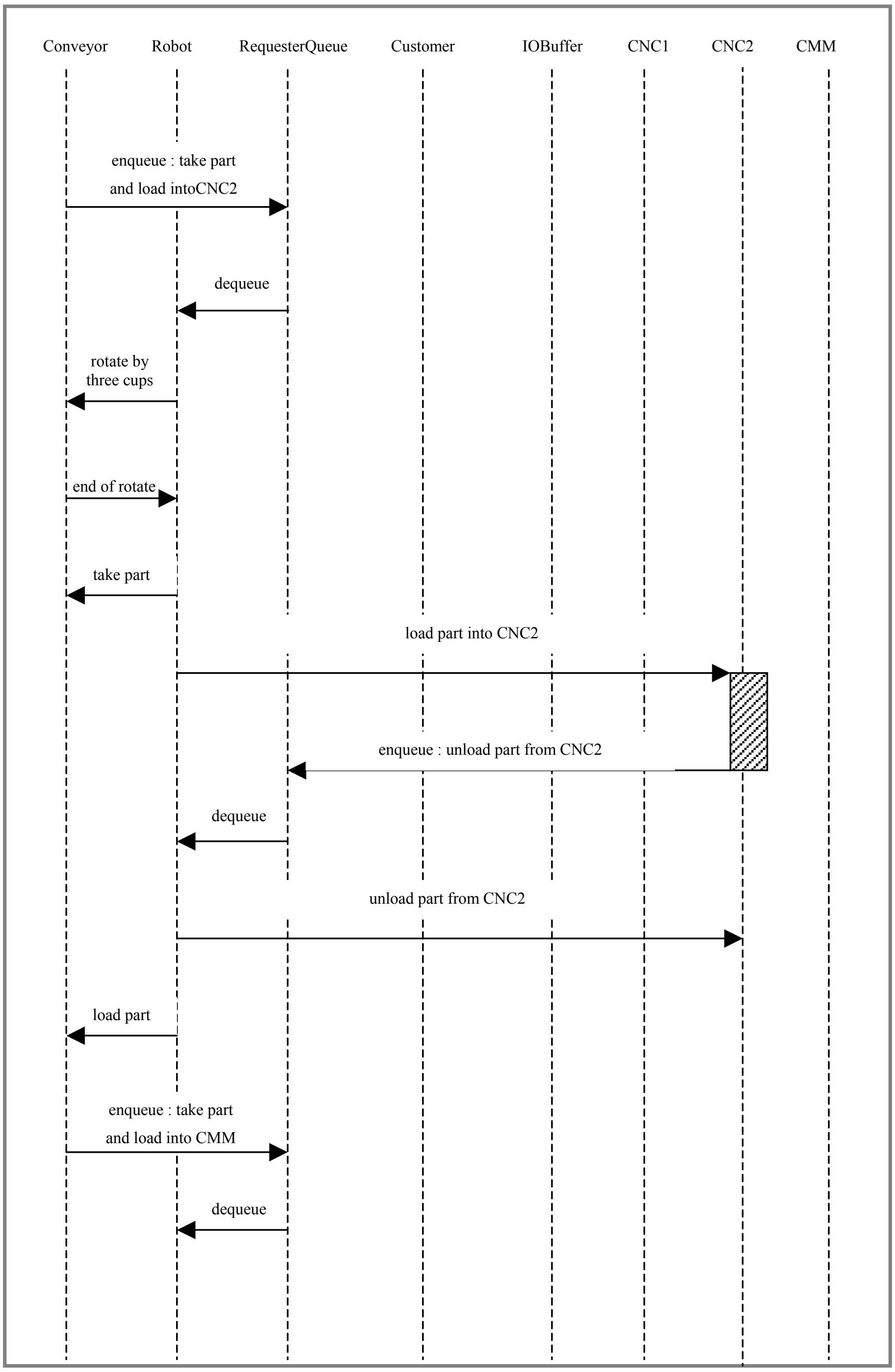

Figure 3.5 Sequence Diagram of Part Type 1 (continued)

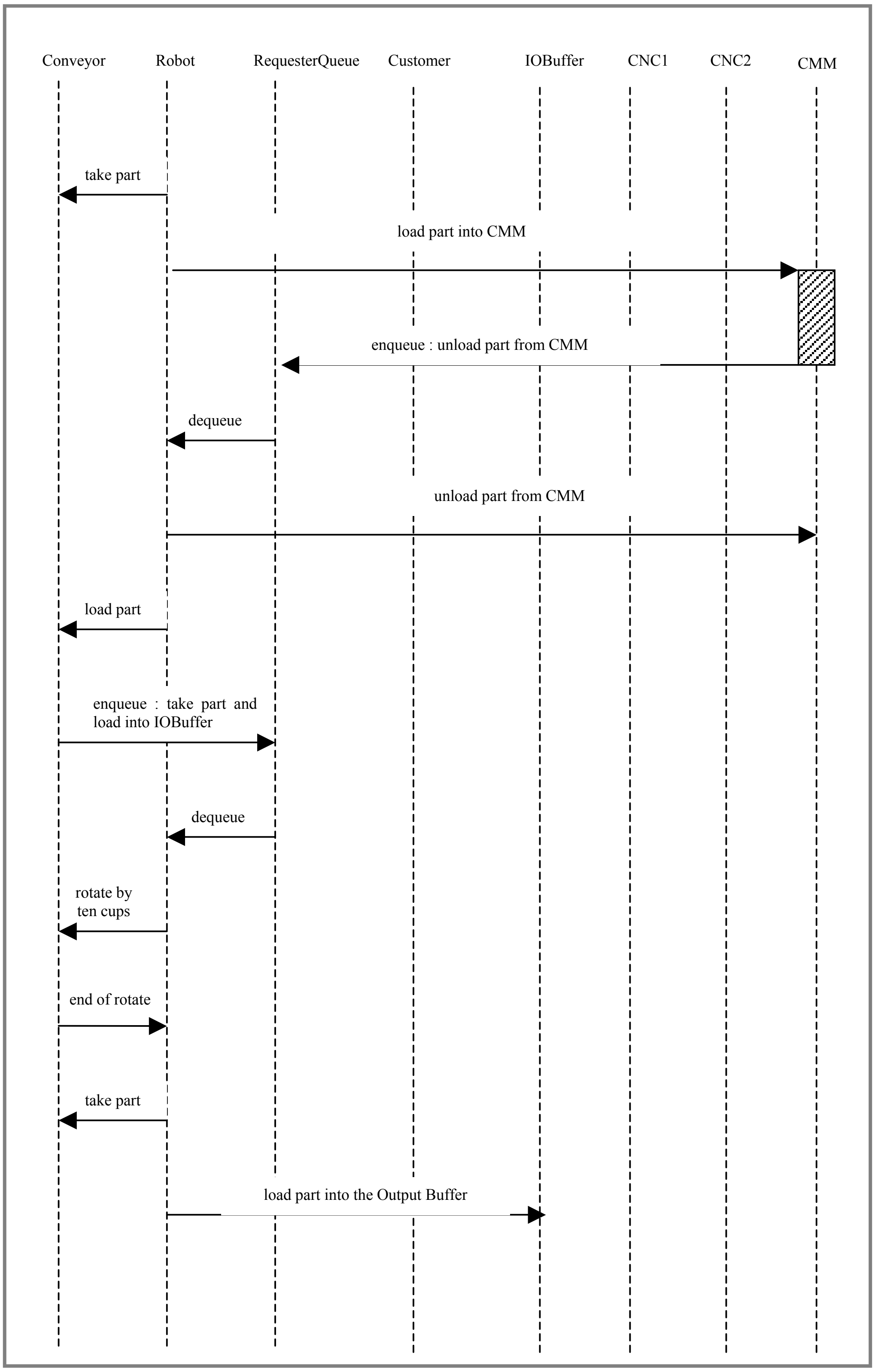

Figure 3.5 Sequence Diagram of Part Type 1 (continued)

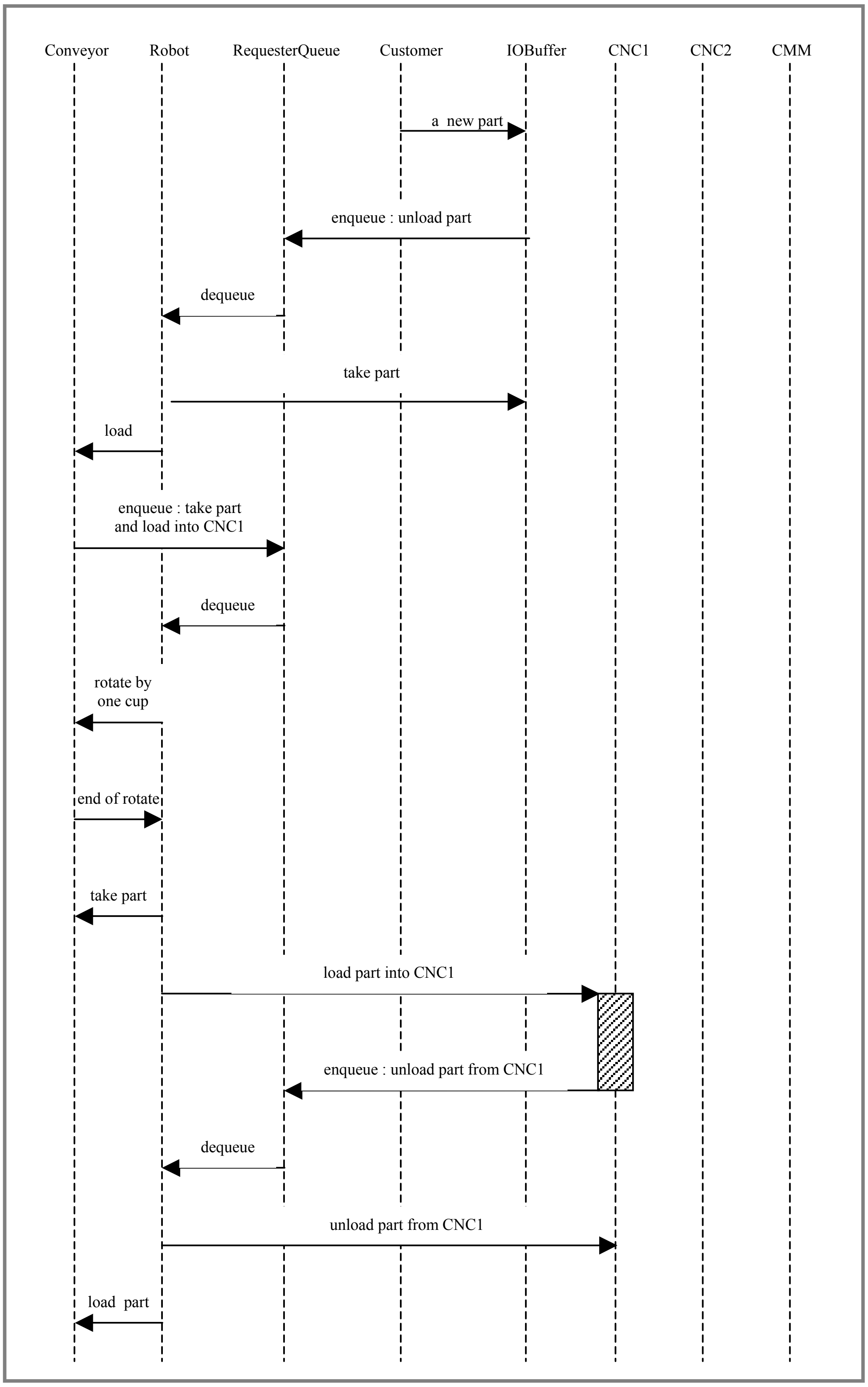

Figure 3.6 Sequence Diagram of Part Type 2

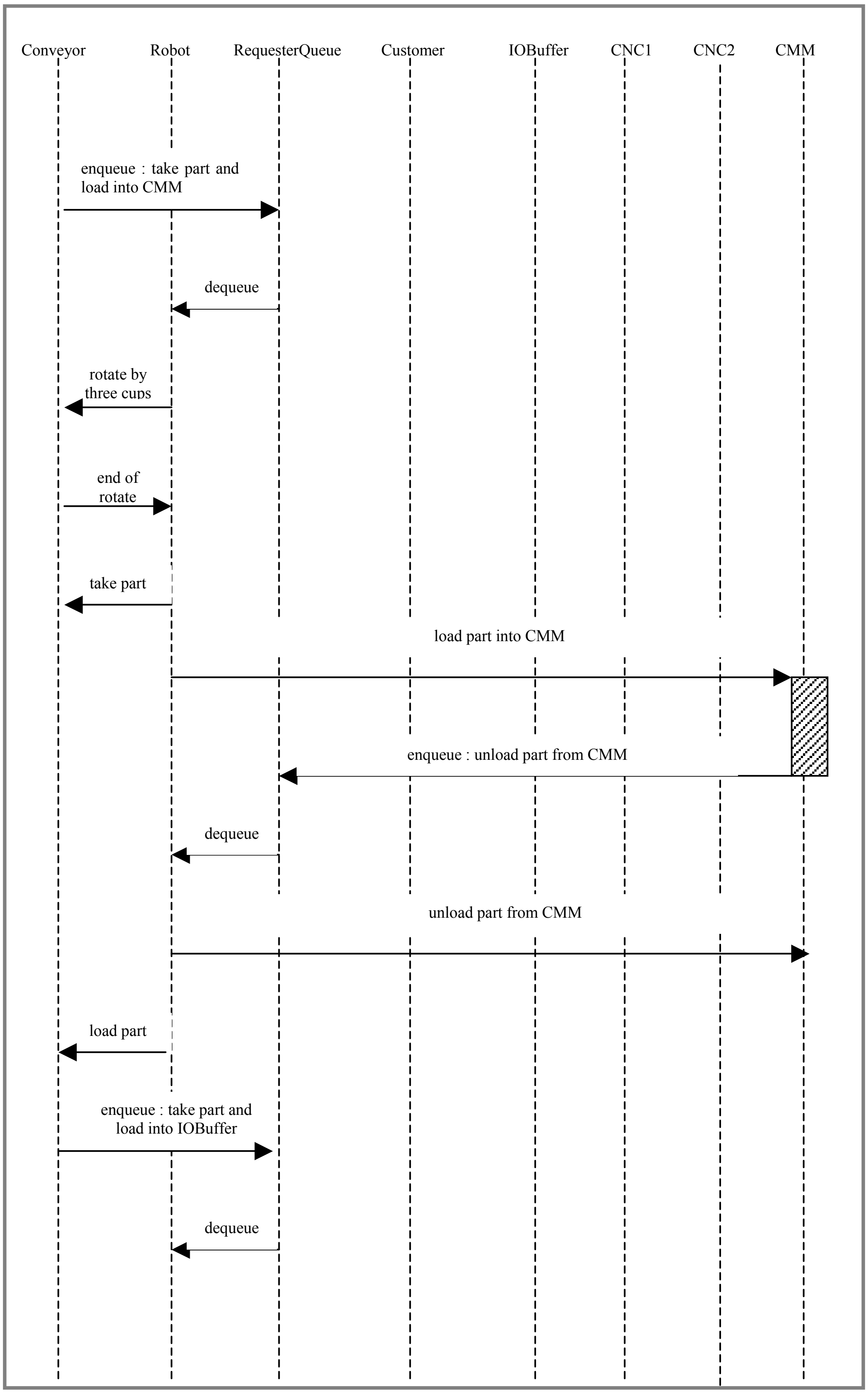

Figure 3.6 Sequence Diagram of Part Type 2 (continued)

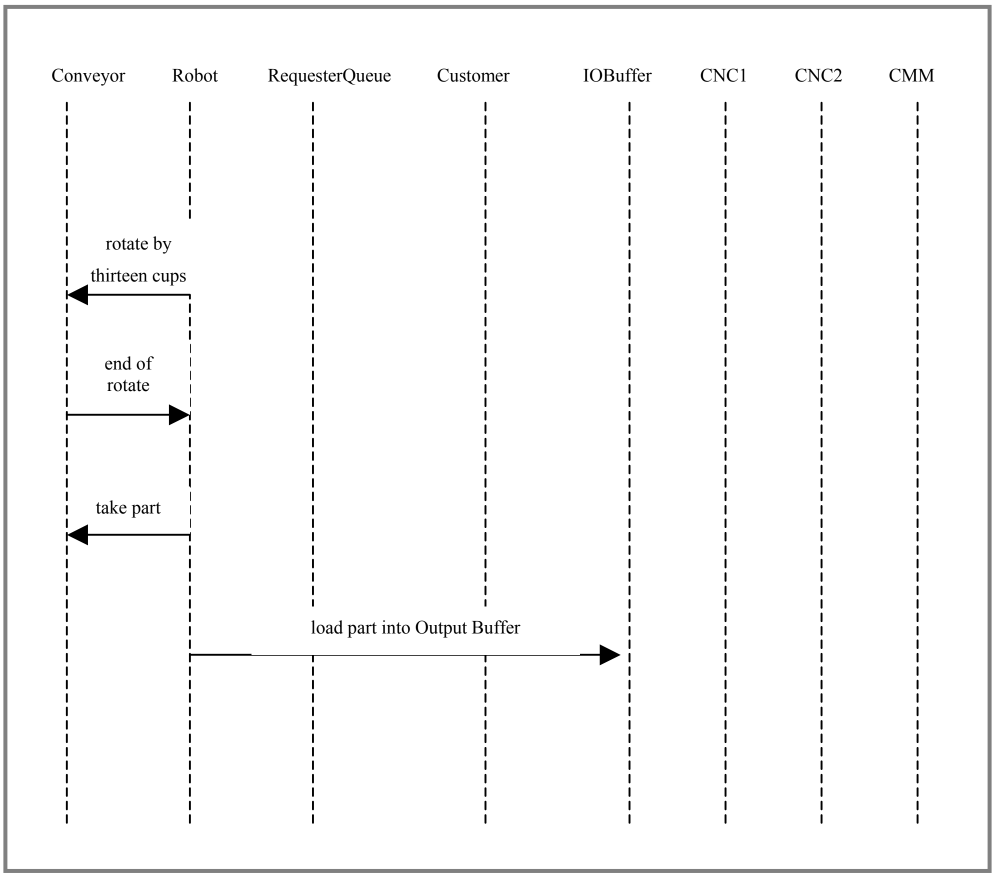

Figure 3.6 Sequence Diagram of Part Type 2 (continued)

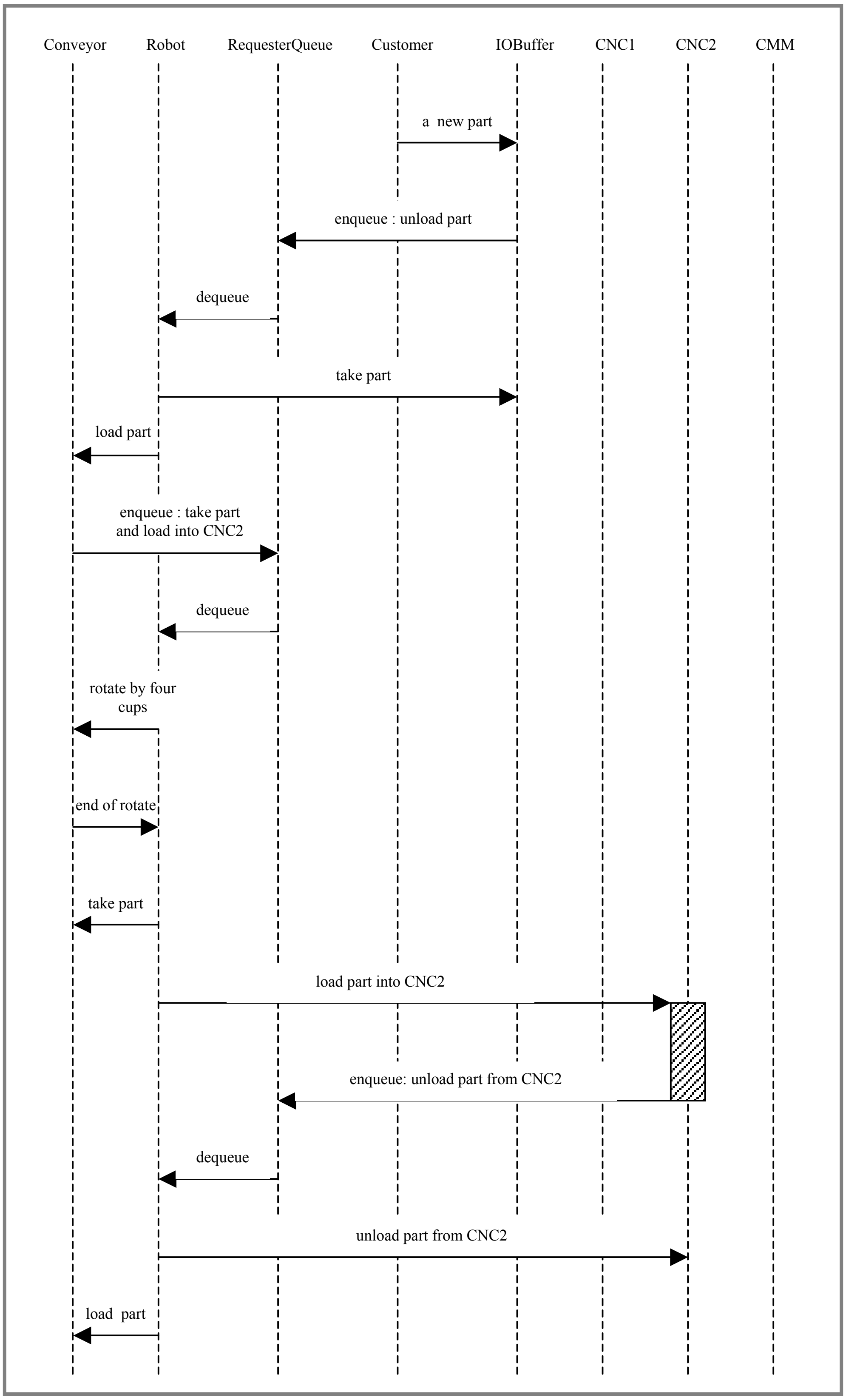

Figure 3.7 Sequence Diagram of Part Type 3

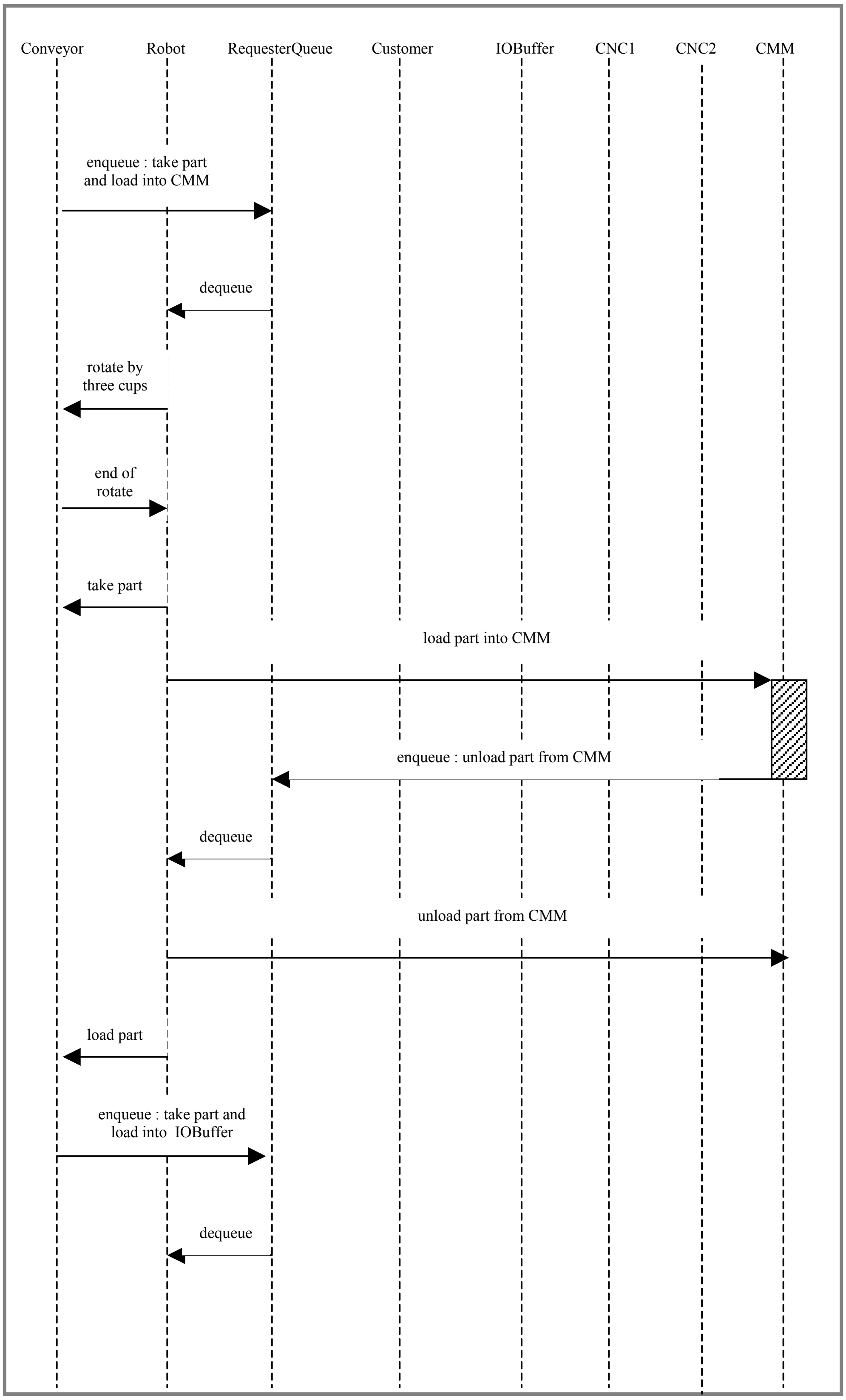

Figure 3.7 Sequence Diagram of Part Type 3 (continued)

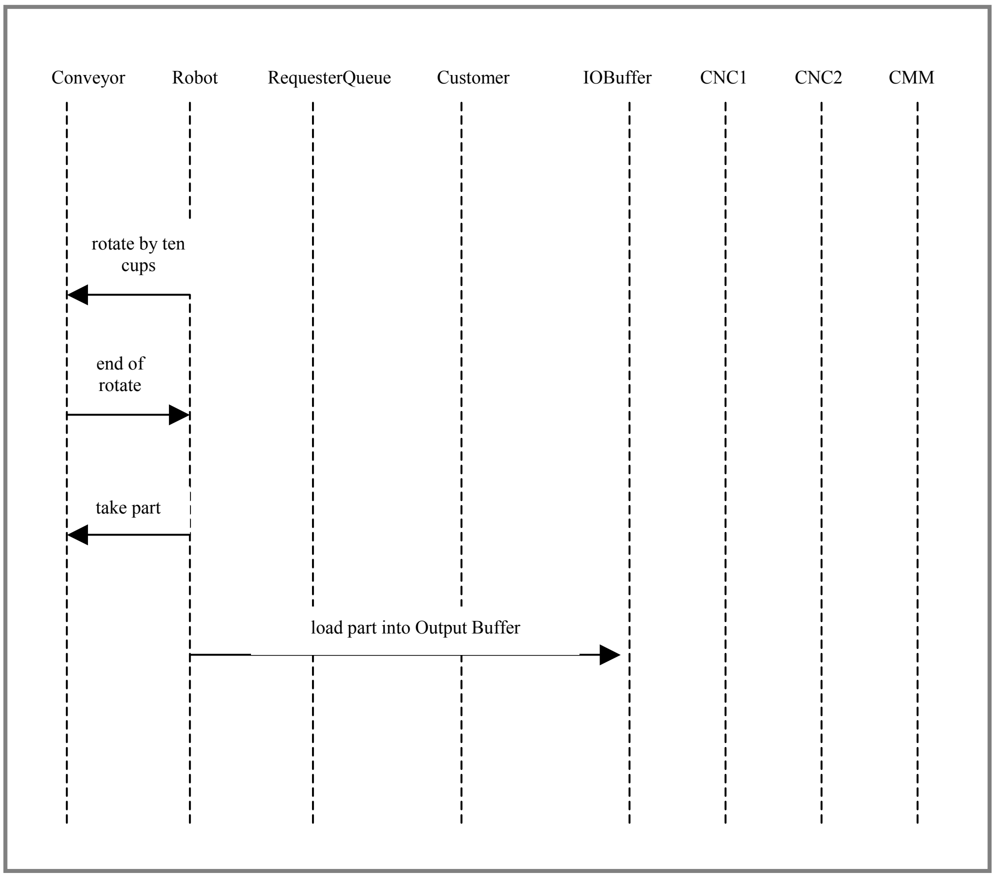

Figure 3.7 Sequence Diagram of Part Type 3 (continued)

CHAPTER 4

VALIDATION STUDY

According to the MaC architecture, requirement specifications have been produced as a result of analyzing the structure and operations of the cell at METUCIM Lab. Our aim is to compare the simulated model with the real world via these requirements, which are extracted from the real system. And again according to the MaC framework, in system design phase, the requirements are also changed into low level.

4.1 *Validation Requirements*

Validation requirements of the METUCIM Lab have been defined group by group to eliminate the repetition. In this way, there are 10 groups of requirements for validating a single run. These are explained in more detail below:

4.1.1 Group 1: Preservation of Parts

There must be at most 14 parts in process at a time. This is because of the limited capacity of the conveyor; it has only 14 cups. Also, the exiting part count of the system must be less than or equal to the entering part count. And, the difference of these two must be equal to the current part count of the system. While the current part count of the system can be found in this way, there is another way to control it: the sum of all the parts in the machines, robot and conveyor must give the existing part count of the system. At the end of the simulation run, the entering part count must be equal to the leaving part count.

This group has 5 validation requirements, three of which are safety properties, and the other two are defined as alarms. The first one checks if the count of entering parts is greater than or equal to the count of exiting parts. In the second one, the difference of the incoming part count from the leaving part count must always give the existing part count. The third one checks if the current part count, which is evaluated according to the entering part and exiting part events, is equal to the total part count in the machines, at the mount of the robot and on the conveyor. The forth one gives alarm if its defined event occurs, which is that at the end of the simulation run, the entering part count is not equal to exiting part count. In fact, it must be equal, all entering parts must exit the system at the end. Therefore, if the event checker catches this event, MaC issues an alarm. The last one is again an alarm, but which should not be hold during the run. In this one, the capacity of the system, which must not exceed 14, is controlled.

1. property *part_count_safe1*
2. property *part_count_safe2*
3. property *part_count_safe3*
4. alarm *part_out_count_error*
5. alarm *capasity_exceeded*

Also, this group has other safety properties and alarms. These are similar to the above ones, but in this time the preservation is checked according to each part type. If there is something wrong with the part count in the run, one of the below violations must occur. However, the same is not true for the above ones.

1. property *part_count_safe1_type1* (for part type 1)
2. property *part_count_safe1_type2* (for part type 2)
3. property *part_count_safe1_type3* (for part type 3)

4. property *part_count_safe2_type1* (for part type 1)
5. property *part_count_safe2_type2* (for part type 2)
6. property *part_count_safe2_type3* (for part type 3)
7. property *part_count_safe3_type1* (for part type 1)
8. property *part_count_safe3_type2* (for part type 2)
9. property *part_count_safe3_type3* (for part type 3)
10. alarm *part_out_count_error1* (for part type 1)
11. alarm *part_out_count_error2* (for part type 2)
12. alarm *part_out_count_error3* (for part type 3)

4.1.2 Group 2: Request-Process Consistency

Eight processes in the system have been focused on at the design phase. While defining them, the movement of the robot and its position are taken as a start point. The first four are defined according to the first position of the robot, which is at the CNC1, while the second four are defined according to the second position of it, which is at the CNC2. The processes are also used for the requests of the system. The requests are done according to these processes.

The requests are evaluated by the robot and if there is something wrong to process them, system gives at least one of the below alarms. According to the name of each implied, they control the eight defined processes. These alarms depend on the events that are monitored in the event recognizer part of the MaC.

1. alarm *wrong_IOBUFFER_process_alarm*
2. alarm *wrong_CNC1_process_alarm*
3. alarm *wrong_CNC2_process_alarm*
4. alarm *wrong_CMM_process_alarm*
5. alarm *wrong_CONVEYOR_IOBUFFER_process_alarm*
6. alarm *wrong_CONVEYOR_CNC1_process_alarm*
7. alarm *wrong_CONVEYOR_CNC2_process_alarm*
8. alarm *wrong_CONVEYOR_CMM_process_alarm*

4.1.3 Group 3: Part Routes Consistency

There are 3 part types of the system and their routes in the system are shown clearly by using their sequence diagrams given in the previous chapter. These part types are chosen according to the number of CNC machines of the system. In METUCIM Lab, there are two such machines. Therefore, we have three part types and three part routes to be checked. These are controlled by the alarms listed below. The events used to describe the alarm are defined in the event recognition part of the system.

1. alarm *wrong_part1_request*
2. alarm *wrong_part2_request*
3. alarm *wrong_part3_request*

4.1.4 Group 4: Limited hardware capacity

All hardware of the system has a limited capacity. Robot can hold only one part at the same time. CNC1, CNC2 and CMM machines process only one part. Finally, the conveyor has only 14 cups, and each cup can carry only one part. Because of the MaC's limitations, which will be described in the following part, entries of the conveyor cannot be monitored one by one. Therefore, only the specific three entries are controlled, which are IOBuffer entry, CNC1 entry and CNC2 entry. As a result, if there is a capacity overflow problem, one of the below alarms is given by the event checker.

1. alarm *robot_capacity_exceeded*
2. alarm *cnc1_capacity_exceeded*
3. alarm *cnc2_capacity_exceeded*
4. alarm *cmm_capacity_exceeded*
5. alarm *conveyor_capacity_exceeded*
6. alarm *iobuffer_entry_capacity_exceeded*
7. alarm *cnc1_entry_capacity_exceeded*
8. alarm *cnc2_enrty_capacity_exceeded*

4.1.5 Group 5: Already empty hardware

If robot tries to unload a part from an already empty cup, or machine, the system gives one of the alarms below. At that point, the event checker catches the unloading event, and decreases the part count of that hardware by one. Therefore, the part count of the hardware starts to have a minus value, which must not happen while the system is running. It can have values only between 0 and the maximum capacity of that hardware. The maximum capacity exceeded control is also done by the previous group.

1. alarm *robot_process_error*
2. alarm *cnc1_process_error*
3. alarm *cnc2_process_error*
4. alarm *cmm_process_error*
5. alarm *conveyor_process_error*
6. alarm *iobuffer_entry_already_empty*
7. alarm *cnc1_entry_already_empty*
8. alarm *cnc2_entry_already_empty*

4.1.6 Group 6: Visitation Consistency

This group is defined to control the visitation consistency of the three machines, which are CNC1, CNC2 and CMM. Incoming parts of the system has three types. According to their types, the total visitation number of the machines can be evaluated. At the end of the simulation run, if these machines are really used as that number then this means everything is satisfied according to this requirement. But, if there is something wrong and the evaluated total is not equal to the total working count then system gives at least one of the alarms below. The alarms are defined as the above logic. When a new part enters the system, the event checker calculates how many times the machines will be used. Also, it keeps other variables to calculate how many times the machines really have been used. At the end of the run, it compares these two values, which are expected to be equal.

1. alarm CNC1_visitation_inconsistent
2. alarm CNC2_visitation_inconsistent
3. alarm CMM_visitation_inconsistent

4.1.7 Group 7: Time Consistency

Like group 6, the working time of the machines, which are CNC1, CNC2 and CMM, must be consistent at the end of the run. If there is a new incoming part, the system evaluates how long the machines will be used. And it also controls and keeps the working duration of those machines while system is running. At the end, it compares these values to check inconsistency. The difference of this group from the other inconsistency checking ones, there is an error in these calculations because of using the system clock for simulation time. Therefore, the below alarms are given if the working duration of the machines exceeds the error bounds. According to the many runs, the error bound is found as approximately ±5% for CNC1 and CNC2, and for CMM, it is found as approximately ±10%, since CMM processes more parts then the others so the error fraction is relatively larger.

1. alarm *cnc1_time_problem*
2. alarm *cnc2_time_problem*
3. alarm *cmm_time_problem*

4.1.8 Group 8: Conveyor Rotation

If conveyor turns by more than 13 cups then this will be an useless rotate, because it has only 13 cups. If an order comes to rotate, but no rotate is needed, then this is also an useless order for rotate. Also, if there is something wrong and rotating order comes with a minus rotate number, then this situation must be hold by the event checker. For all of these cases, an alarm is given by the system, which is copied below.

1. alarm *conveyor_rotate_problem*

4.1.9 Group 9: Robot sides

The processes, which are also the requests of the system, have been defined according to the robot movements and the position of the robot, which is mentioned above. There are two position of the robot in this manufacturing cell. Four processes must be done in the first position, and the other second must be done at the second side. If the robot seems operating at the wrong side, the below alarm is given by the system.

1. alarm *robot_at_wrong_side*

4.1.10 Group 10: Minimum time between two consecutive exits

If any manufactured part exits from the system, then the second exiting part must leave the system after minimum 30000ms. This is the total time of the following sequential works:

1. Robot gets its normal position after putting the first part onto the output buffer.

2. The request for the second part comes to the robot and robot wants this part from the conveyor. Conveyor rotates by one cup (the second part must be on the next cup of the first one to calculate the minimum time).

3. Robot takes the part from the conveyor.

4. Robot puts the second part onto the output buffer.

The defined alarm for this group is below:

1. alarm *min_time_difference*

4.2 *MaC Scripts*

According to the MaC framework, the validation requirements are defined while the system requirements are worked on. After the design phase of the simulation development, these high-level requirements are changed into low-level information by using the variables of the designed system.

First, the primitive-event definitions are written, this is the only script, which communicates with the simulation run. The variables of the simulation code can only be seen by this script, so in this one, most of the events and conditions depending on the different state of the object attributes can be defined.

The second script, written by using the meta-event definition language, imports the necessary events and conditions. In this script, auxiliary variables can be defined and are used to keep historical information about the system. New events and conditions can be defined by using these auxiliary variables, imported events and conditions. The conditions are used to define the safety conditions of the system, and the events are used to define the alarms.

4.2.1 Limitations and Problems with MaC Tool

While writing and testing the scripts, lots of problems and limitations were encountered such that sometimes finding the source of the error took one month, sometimes it was impossible to find the error, so other ways, which were not practical, were followed to get around it. The problems and limitations are written below one by one, with necessary explanations, or examples.

1. In the PEDL script, an array entry cannot be monitored. Because of this limitation, the conveyor array of the simulation code cannot be written in this script to be monitored. To get around this limitation, three same classes with different names have been defined (the answer of “why not one class” is another item of the problems). Each of them is corresponding to the each three specific entry of the conveyor. The name of these classes are:

- **a.** IOBUFFER_Cup_Entry
- **b.** CNC1_Cup_Entry
- **c.** CNC2_Cup_Entry

When an assignment is done to one of the three entries of the conveyor, it is also done to the corresponding class instant. Consequently, in the PEDL script, these three classes are monitored instead of the conveyor itself.

2. At the beginning of the work, experience with MaC was done with programs that had one or two classes some of which had threads. Also, some changes were done on the examples of the MaC's itself, and they were tried to run, and success was obtained. After writing the first codes of METUCIM Lab simulation with 10 classes, and a small script to only see the monitoring was ok, the event recognizer of MaC gave run-time errors. Several things done; at the end, "return;" was added at the end of all void functions and it started to work. However, it is so misleading that for the programs with one or two classes, it does not require such a thing.

3. It does not allow use of named constants. Therefore, instead of constants, literals are used. For example, for the request named by IOBUFFER, the literal value, 1, is used.

```
condition IOBUFFER = (Robot.x_request_type == 1);
```

4. The conditions and the events were written group by group, each group was tested before to sure they were correct. Some strange things happened again. When a group of three conditions are defined, it gave run-time error again. These conditions are:

```
condition part_cnt_control1 = (METUCIM.val_part_cnt_control == 0) ||
                       ((Robot.val_full + Cnc1.val_full + Cnc2.val_full +
                        Cmm.val_full   + Conveyor.val_full) <= 14);

condition part_cnt_control2 = ((METUCIM.val_part_cnt_control == 0) ||
                       ((Robot.val_full + Cnc1.val_full + Cnc2.val_full +
                        Cmm.val_full   + Conveyor.val_full) ==
(METUCIM.val_count_enter_parts - METUCIM.val_count_exit_parts)));

condition part_cnt_control3 = METUCIM.val_count_enter_parts >=
                                    METUCIM.val_count_exit_parts;
```

After several tries, it was found out that, the source of the run-time error is that the condition 2 and condition 3 could not written together, so the condition 3 was commented out and rewritten at the back of condition two by using “&&”.

At first, the source of the error can be seen that using same monitored variables can cause a run-time error. But, this is not true, anyway condition 1 and 2 use same variables, also when the below condition was added which liked the third one, the recognizer gave no error:

```
condition part_cnt_control4 =  (METUCIM.val_count_enter_parts –
                                  METUCIM.val_count_exit_parts) >

      METUCIM.CONVEYOR_CUP_NUMBER;
```

But the problem in this case was the event checker. Although, while the condition 4 was tried by alone, there was no run-time error, when it was used together with the condition 1 and 2, event checker gave run-time error. This problem was managed (after found out) in a same way such that the condition 4 was also commented out and added to the back of condition 1 with “&&”.

Also, note that, these ones are not used the last version of the scripts. The controls done by using them is now the function of the events defined in MEDL script, by the help of the auxiliary variables.

5. When the input buffer size was increased from 200 to 2000, event checker gave run-time error.

6. Inheritance cannot be used. When an attribute of such a class is wanted to be monitored, the event recognizer does not see it from the child class, or the parent class.

While defining the conditions, if end(*condition*) statement is used then one must be more careful. This statement means that when the *condition* becomes false, this is caught by using *end* predefined function. The problem is that, at beginning of the simulation run, this *condition* is undefined and it catches this event. When a predefined function definition is used, like end(defined(*condition*)), it still behaves in the same way.

7. After importing lots of events from PEDL script, MEDL script started not use the imported conditions. Importing could be done, but when the imported conditions were tried to be used in MEDL script, event checker gave run-time error.

8. It did not see a local variable of a method while instrumenting the class of this method. Before that, at the first experience of MaC, such a thing could be done. While the code of the script gets large, it may give these errors.

9. If the simulation classes are instrumented, the daemon property of the thread classes does not do their jobs, which are that when the main function ends, if this property has been set, the thread must end also.

4.2.2 Primitive Event Definition Script of the Simulation

The primitive event definition script is in Appendix A. In this section, the monitored objects and methods are explained.

1. Monitored Attributes:

METUCIM.val_part_cnt_control : This attribute helps control the time of transferring a part between the robot and the hardware. When the value of this attribute is not 1, the part count controls are not done. Since at the time of getting or putting a part, it cannot be determined to which hardware the part belongs, because of the two consecutive assignment statements, first of which makes the robot full or empty. If this control is not in effect at that time then the part cannot be seen in the system, or it can be seen as double.

METUCIM.end_threads : By using this monitored object, the end of the run can be detected.

Robot.val_full, Cnc1.val_full, Cnc2.val_full, Cmm.val_full : These attributes are used to check that hardware is full or empty.

Cnc1.part_type, Cnc2.part_type, Cmm.part_type : To get the type of the part inside these machines, these attributes are monitored.

Cnc1.start_work, Cnc2.start_work, Cmm.start_work : By these objects, how long these machines process the parts can be obtained.

IOBUFFER_Cup_Entry.val_full, CNC1_Cup_Entry.val_full, CNC2_Cup_Entry.val_full : These classes are created to monitor the three specific entry index of the conveyor. These attributes show that whether the entry is empty or not.

IOBUFFER_Cup_Entry.part_type,
CNC1_Cup_Entry.part_type,
CNC2_Cup_Entry.part_type : Also, knowing the type of the parts on these entries is important.

Robot.x_request_type, Robot.x_part_type, Robot.x_from : These are attributes of the request, which is deleted by the robot from the requester queue to be processed. Robot will operate according to the request type, so it wants the first matching part type from the system. Also, the system must know where the part has been lastly; because there can be several parts of same type, which are not at that step of their manufacturing processes.

Robot.after_turn : To watch the steps of the robot process, this attribute is used. Before dequeueing the first request, its value is 0. Then, the part is made ready for taking by the robot. Now, its value is 1. After robot takes the part, it is 2. Finally, after robot puts the part, it is 3.

Robot.robot_place : To track the robot movement on the phenoumetic buffer, this attribute is monitored.

Conveyor.turn_number : This is monitored to check whether there is an abnormal rotate of conveyor or not.

Conveyor.turn_flag : If this flag is set, this means that the conveyor rotates.

IOBuffer.first_element : Robot can always take the first element of the input buffer. This attribute value is controlled for this requirement.

2. Monitored Methods:

METUCIM.main(String[]) : Start of the simulation run is important to set all auxiliary variables of MEDL script. Therefore, start of this method is monitored.

4.2.3 Meta Event Definition Script of the Simulation

In this script, the alarms, which must never raise, and the safe conditions, which must always hold true, are defined. The necessary events, which depend on the monitoring object values, are imported. There is no imported conditions since there is a problem in the MaC tool with using them. The meta event definition script of this simulation is at Appendix B.

4.3 *Fault Injected Simulations*

After the METUCIM Lab simulation has been completed, series of test runs have been performed according to the specified requirements told above. MaC tool has not given any error against the validation rules written in the script. By being sure of that this is the true simulation code according to the wanted requirements, it is rewritten to test the validation groups. In total, 27 scenarios have been worked on. The scenarios that have been though as if someone misunderstood the requirements, or coded something wrong. Afterwards, the true simulation code is changed into its fault injected versions, the count of which are 47. These are told below one by one with necessary explanations. First the scenario is described. Afterwards, expected alarms and violated properties are written before the real alarms and violated properties, which are given by MaC while running this fault injected simulations. There can be differences between them. In some cases, there can be extra violations, of whose occurrences are impossible, given by the system beside the expected ones, like the compiler errors, and they might be called “superfluous alarms”. In some cases, even expected violations are not raised by the checker because the system cannot reach that state. The reasons of these are explained at the end of each scenario.

4.3.1 Scenario 1: Using IOBuffer entry instead of CNC1 entry

CNC1 and the IOBuffer entries are two consecutive cups on the conveyor. In this scenario, CNC1 entry was ignored and all jobs using this entry were rewritten to use IOBuffer entry instead.

Example for the Violation of Requirement Groups:

1. Group 2 : Request-Process Consistency

After execution:

1. wrong_CONVEYOR_CNC1_process_alarm *(2.group)*
2. wrong_CNC1_process_alarm *(2.group)*
3. part_count_safe3 *(1.group)*
4. part_count_safe3_type2 *(1.group)*
5. part_count_safe3_type1 *(1.group)*
6. conveyor_capacity_exceeded *(4.group)*

Although the third and the forth alarms were not expected, they were given because of the monitor does not see the from_conveyor event which is defined according to the emptiness of the entries. In this case, CNC1 entry is still full, while robot takes the part from IOBuffer entry. Therefore, from_conveyor event does not work. However, while putting the part onto IOBuffer entry, because the CNC1 entry is full, it recognizes onto_conveyor event. This means that there is part count inconsistency in the run, and finally, the conveyor capacity is greater than 14.

4.3.2 Scenario 2: Directly loading parts into CMM from CNC2

The parts are loaded into the machines or iobuffer from always conveyor. No directly transfer is allowed from one machine to another. In this scenario, this requirement is violated such that robot loads the part into CMM after taking it from CNC2.

Example for the Violation of Requirement Groups:

1. Group 2 : Request-Process Consistency

After execution:

1. wrong_CNC2_process_alarm *(2.group)*
2. part_count_safe3 *(1.group)*
3. part_count_safe3_type1 *(1.group)*
4. part_count_safe3_type3 *(1.group)*
5. conveyor_capacity_exceeded *(4.group)*

The explanation of the unexpected violations is that while the part is loaded into CMM, if the CNC2 entry is empty then the event recognizer sees this as onto_conveyor event. However, since CONVEYOR_CMM request never comes, the corresponding from_conveyor event does not occur. Therefore, when CNC2 request comes to the cell, if its entry is empty then the part count on the conveyor always increases, and as a result, event recognizer gives the second and the third violations. The reason that the second expected alarm does not occur is that the CONVEYOR_CMM request never comes to the robot because the job as a result of this request is done by CNC2 request.

4.3.3 Scenario 3: Parts are not measured

All parts are measured by CMM before leaving the system. In this scenario, the parts exit the system without loading into CMM.

Example for the Violation of Requirement Groups:

1. Group 3 : Part Routes Consistency
2. Group 6 : Visitation Consistency
3. Group 7 : Time Consistency

After execution:

1. wrong_ part1_request *(3.group)*
2. wrong_ part2_request *(3.group)*
3. wrong_ part3_request *(3.group)*
4. cmm_visitation_inconsistency *(6.group)*
5. cmm_time_problem *(7.group)*

All expected alarms are issued and no superfluous alarms are given in this test. All of the third group violations occur since all routes are wrong and at the end of the simulation run, the CMM visitation is inconsistent.

4.3.4 Scenario 4: Loading second part into the machines

The capacities of the machines are limited by 1. This means that there can be only one part at the same time to be manufactured or the machine must be empty. In this scenario, the robot tries to load second parts while the machines are already full.

Example for the Violation of Requirement Groups:

1. Group 2 : Request-Process Consistency
2. Group 4 : Limited Hardware Capacity

After execution:

1. wrong_CONVEYOR_CNC1_process_alarm *(2.group)*
2. wrong_CNC1_process_alarm *(2.group)*
3. cnc1_capacity_exceeded (4.group)
4. wrong_CONVEYOR_CNC2_process_alarm *(2.group)*
5. wrong_CNC2_process_alarm *(2.group)*
6. cnc2_capacity_exceeded (4.group)
7. wrong_CONVEYOR_CMM_process_alarm *(2.group)*
8. wrong_CMM_process_alarm *(2.group)*
9. cmm_capacity_exceeded (4.group)

The simulation does not end because when robot loads second part into any machine, the previous part is erased from the system according to this fault-injected simulation. Therefore, the expected violations at end of the simulation do not occur. Also, the reason to take the process errors although there seems no process error is that, the checker controls the part type from which the request comes. If it is not same as the part type in the machine, the process alarms of group 2 raise.

The interesting thing in this scenario is that while the parts get lost in the system, the last expected violation does not occur. This is because of the limited defined events for event recognizer. At the design phase, handling all faults like this scenario is impossible, so the events are defined according to the normal running situation of the system. Therefore, by the guide of the assumption, no part gets lost in the system, machines' part counts are always increased, and everything seems proper to that safety property, although the machines' capacities are exceeded. Anyway, this is caught by the violation group 4.

4.3.5 Scenario 5: Robot drops the parts

The parts in the system should not get lost while simulation running. In this scenario, this requirement violation is tested again. While the robot loading or unloading a part, it drops the part. This scenario is detailed one by one for each machine and the conveyor.

For this test group, the general expected violations controlled at the end of the simulation are written below and the explanation about them will be done here.

1. Group 1 : Preservation of Parts
2. Group 6 : Visitation Consistency
3. Group 7 : Time Consistency

None of the simulations of 5th scenario ends its run, because of losing the parts in the cell. The cell does not end until it manufactures all entered parts. Therefore, the simulations described below do not reach at these violation states.

1. Robot drops the part while loading into CNC1:

Example for the Violation of Requirement Groups:

1. Group 1 : Preservation of Parts
2. Group 2 : Request-Process Consistency

After execution:

1. wrong_CONVEYOR_CNC1_process_alarm *(2.group)*
2. part_count_safe3 *(1.group)*
3. part_count_safe3_type1 *(1.group)*
4. part_count_safe3_type2 *(1.group)*

For the first case, expected violations, except from the general ones, occur. Robot drops the part while loading, so CNC1 does not know that there is a part wanted to be loaded into itself. Everything goes on as nothing happens, but only the current part count of the system decreases which also causes violation.

2. Robot drops the part while loading into CNC2:

Example for the Violation of Requirement Groups:

1. Group 1 : Preservation of Parts
2. Group 2 : Request-Process Consistency

After execution:

1. wrong_CONVEYOR_CNC2_process_alarm *(2.group)*
2. part_count_safe3 *(1.group)*

3. part_count_safe3_type1 *(1.group)*
4. part_count_safe3_type3 *(1.group)*

The same things described for the first one is valid for this test case. In this case, CNC2 does not know that there is a part, which is to be loaded into itself. The part is dropped and CNC2 does not sense anything. However, the existing part count is not equal to the part count calculated according to the incoming and outgoing part numbers.

3. Robot drops the part while loading into CMM.

Example for the Violation of Requirement Groups:

1. Group 1 : Preservation of Parts
2. Group 2 : Request-Process Consistency

After execution:

1. wrong_CONVEYOR_CMM_process_alarm *(2.group)*
2. part_count_safe3 *(1. group)*
3. part_count_safe3_type1 *(1.group)*
4. part_count_safe3_type2 *(1.group)*
5. part_count_safe3_type3 *(1.group)*

The first three fault injections are same as each other, so the results and the explanation of those results are almost same. In this case, robot drops the part while putting into CMM and no hardware of the system is effected from this.

4. Robot drops the part while loading onto conveyor:

Example for the Violation of Requirement Groups:

1. Group 1 : Preservation of Parts
2. Group 2 : Request-Process Consistency

After execution:

1. wrong_IOBUFFER_process_alarm *(2.group)*
2. wrong_CNC1_process_alarm *(2.group)*
3. wrong_CNC2_process_alarm *(2.group)*
4. wrong_CMM_process_alarm *(2.group)*
5. wrong_CONVEYOR_CNC1_process_alarm *(2.group)*
6. wrong_CONVEYOR_CNC2_process_alarm *(2.group)*
7. wrong_CONVEYOR_CMM_process_alarm *(2.group)*
8. iobuffer_entry_already_empty *(5.group)*
9. cnc1_entry_already_empty *(5.group)*
10. cnc2_entry_already_empty *(5.group)*
11. conveyor_process_error *(5.group)*
12. part_count_safe3 *(1.group)*
13. part_count_safe3_type1 *(1.group)*
14. part_count_safe3_type2 *(1.group)*
15. part_count_safe3_type3 *(1.group)*

This case is different from the first three since for this fault it is assumed that the conveyor does not sense whether there is a part or not on its specific cup. Robot cannot load the part, but it suggests that the part has been put, so the requests have been done for that dropped part and everything gets mixed while the run goes on. The difference is that the machines sense whether there is a part inside itself or not, so according to this they start to work, however, conveyor is controlled by the robot. If robot does a wrong thing and does not notice like this situation, conveyor cannot handle this error.

5. Robot drops the part while unloading from CNC1

Example for the Violation of Requirement Groups:

1. Group 1 : Preservation of Parts
2. Group 2 : Request-Process Consistency

After execution:

1. wrong_CNC1_process_alarm *(2.group)*
2. wrong_CONVEYOR_CNC2_process_alarm *(2.group)*
3. wrong_CONVEYOR_CMM_process_alarm *(2.group)*
4. conveyor_process_error *(5.group)*
5. robot_process_error *(5.group)*
6. cnc1_entry_already_empty *(5.group)*
7. cnc2_entry_already_empty *(5.group)*
8. part_count_safe3 *(1.group)*
9. part_count_safe3_type1 *(1.group)*
10. part_count_safe3_type2 *(1.group)*

The main reason of the superfluous alarms are explained at the end of the previous fault injected case. The reason is that the conveyor does not sense whether the part is put onto it successfully or not, so the consecutive alarms raise. The parts are not put onto conveyor, but the system continues to run, and makes requests for this parts. Therefore, although the entries are empty, robot tries to take the part. Also, robot tries to put a part while there is no part at its mount. All of these raise the half of the group 5 alarms.

6. Robot drops the part while unloading from CNC2:

Example for the Violation of Requirement Groups:

1. Group 1 : Preservation of Parts
2. Group 2 : Request-Process Consistency

After execution:

1. wrong_CNC2_process_alarm *(2.group)*
2. wrong_CMM_process_alarm *(2.group)*
3. wrong_CONVEYOR_CMM_process_alarm *(2.group)*
4. conveyor_process_error *(5.group)*
5. robot_process_error *(5.group)*
6. cnc2_entry_already_empty *(5.group)*

7. iobuffer_entry_already_empty *(5.group)*
8. part_count_safe3 *(1.group)*
9. part_count_safe3_type1 *(1.group)*
10. part_count_safe3_type3 *(1.group)*

The same explanation as in fifth one can be done for this case since only the machine is different. However, in this one, the same entry cnc2 entry is used for also loading the CMM, so CMM process is effected from this situation.

7. Robot drops the part while unloading from CMM:

Example for the Violation of Requirement Groups:

1. Group 1 : Preservation of Parts
2. Group 2 : Request-Process Consistency

After execution:

1. wrong_CNC2_process_alarm *(2.group)*
2. wrong_CMM_process_alarm *(2.group)*
3. wrong_CONVEYOR_CMM_process_alarm *(2.group)*
4. conveyor_process_error *(5.group)*
5. robot_process_error *(5.group)*
6. cnc2_entry_already_empty *(5.group)*
7. iobuffer_entry_already_empty *(5.group)*
8. part_count_safe3 *(1.group)*
9. part_count_safe3_type1 *(1.group)*
10. part_count_safe3_type2 *(1.group)*
11. part_count_safe3_type3 *(1.group)*

This is not also different from the above one. Robot uses the same entry for loading or unloading CNC2 and CMM, so both are effected from this case.

8. Robot drops the part while unloading from conveyor

Example for the Violation of Requirement Groups:

1. Group 1 : Preservation of Parts
2. Group 2 : Request-Process Consistency

After execution:

1. wrong_CONVEYOR_CNC1_process_alarm *(2.group)*
2. wrong_CONVEYOR_CNC2_process_alarm *(2.group)*
3. robot_process_error *(5.group)*
4. part_count_safe3 *(1.group)*
5. part_count_safe3_type1 *(1.group)*
6. part_count_safe3_type2 *(1.group)*
7. part_count_safe3_type3 *(1.group)*

In this case, some of the expected violations do not occurred because system losts the parts immediately after they enter the system either unloading to put into CNC1 or CNC2. The parts of types 1 or 2 get lost after CNC1 request, and the parts of type 3 get lost after CNC2 request. They cannot reach to CMM and the output buffer.

4.3.6 Scenario 6: Wrong part routes

There are three part types defined for the system. The process routes of each are certain. In this case, these routes are changed. Three different scenarios are worked on for handle all three paths.

1. The parts of type 1 do not make a request to be manufactured in CNC1:

Example for the Violation of Requirement Groups:

1. Group 3 : Part Routes Consistency
2. Group 6 : Visitation Consistency

3. Group 7 : Time Consistency

After execution:

1. wrong_part1_request *(3.group)*
2. CNC1_visitation_inconsistent *(6.group)*
3. cnc1_time_problem *(7.group)*

The parts of type 1 should firstly processed by CNC1. They skip this machine, so the evaluated visitation number is not consistent after the simulation ends.

2. The parts of type 2 are only manufactured by CNC2:

Example for the Violation of Requirement Groups:

1. Group 3 : Part Routes Consistency
2. Group 6 : Visitation Consistency
3. Group 7 : Time Consistency

After execution:

1. wrong_part2_request *(3.group)*
2. CNC1_visitation_inconsistent *(6.group)*
3. CNC2_visitation_inconsistent *(6.group)*
4. cnc1_time_problem *(7.group)*

The parts of type 2 are only manufactured by CNC1. In such a scenario, CNC1 is omitted from the route, and instead CNC2 is used. Therefore, CNC1 is not used as expected as, while CNC2 is used more than evaluated visitation count. However, for part type 2, CNC2 does not work, it does not know this part type, so the time problem for this machine does not raise.

3. The parts of type 3 are manufactured both by CNC1 and CNC2:

Example for the Violation of Requirement Groups:

1. Group 3 : Part Routes Consistency
2. Group 6 : Visitation Consistency
3. Group 7 : Time Consistency

After execution:

1. wrong_part3_request *(3.group)*
2. CNC1_visitation_inconsistent *(6.group)*

This type of parts are only processed by CNC2. In this case, they also request to be manufactured by CNC1, so CNC1 visitation inconsistency occurs at the end of the simulation run.

4.3.7 Scenario 7: Taking a part from an empty machine

Robot cannot try to take a part from an already empty machine. In this scenario, this requirement is violated. In the simulation code, after each request evaluated by robot, the system automatically creates the proper request according to the machine if that machine is empty.

1. Robot tries to take a part from CNC1 while CNC1 is already empty:

Example for the Violation of Requirement Groups:

1. Group 2 : Request-Process Consistency
2. Group 5 : Already empty hardware

After execution:

1. wrong_IOBUFFER_process_alarm *(2.group)*
2. wrong_CNC1_process_alarm *(2.group)*
3. wrong_CNC2_process_alarm *(2.group)*

4. wrong_CMM_process_alarm *(2.group)*
5. iobuffer_entry_capacity_exceeded (4.group)
6. cnc1_entry_capacity_exceeded (4.group)
7. cnc2_entry_capacity_exceeded (4.group)
8. conveyor_capacity_exceeded (4.group)
9. cnc1_process_error *(5.group)*
10. part_count_safe3 *(1.group)*
11. part_count_safe3_type1 *(1.group)*
12. part_count_safe3_type2 *(1.group)*

The code of evaluating the requests, which come from the machines in the correct simulation code, has not changed. In normal run, this cannot be happen, so the emptiness of the machines are not controlled in the code. However, if it is changed to create this scenario, there are lots of superfluous violations. This is because of that robot does not control whether there is a part in the machine or not, it supposes to take the part and to put it onto the conveyor, and the system continues to run as nothing happens. This is same as the scenario(5.4) of dropping part while putting it onto the conveyor.

2. Robot tries to take a part from CNC2 while CNC2 is already empty:

Example for the Violation of Requirement Groups:

1. Group 2 : Request-Process Consistency
2. Group 5 : Already empty hardware

After execution:

1. wrong_CNC1_process_alarm *(2.group)*
2. wrong_CNC2_process_alarm *(2.group)*
3. cnc1_entry_capacity_exceeded (4.group)
4. cnc2_entry_capacity_exceeded (4.group)
5. conveyor_capacity_exceeded (4.group)
6. cnc2_process_error *(5.group)*

The same explanation of the previous one is valid for this case. However, in this one the count of raised violations is less because the system starts to run incorrectly after the first machine, so the states, which cause the violations, have less time to occur. This reason decreases the number of violation variety.

3. Robot tries to take a part from CMM while CMM is already empty:

Example for the Violation of Requirement Groups:

1. Group 2 : Request-Process Consistency
2. Group 5 : Already empty hardware

After execution:

1. wrong_CMM_process_alarm *(2.group)*
2. part_count_safe3 *(1.group)*
3. part_count_safe3_type1 *(1.group)*
4. part_count_safe3_type2 *(1.group)*
5. part_count_safe3_type3 *(1.group)*
6. conveyor_capacity_exceeded (4.group)
7. cmm_process_error *(5.group)*

The decrease in the violation count continues in this case, because CMM is the last machine. This scenario has less time to reach the violated states than the previous scenario.

None of the simulations has ended in this scenario because of the improper work of the system and as a result of meaningless requests.

4.3.8 Scenario 8: Taking a part from an empty cup

Robot cannot try to take a part from an already empty cup of the conveyor. This scenario looks like the previous one. But, in this one the violations raised are different. To handle them separately, this is handled as a new scenario.

1. Robot tries to take a part from IOBUFFER entry while IOBUFFER entry is already empty:

Example for the Violation of Requirement Groups:

1. Group 2 : Request-Process Consistency
2. Group 5 : Already empty hardware

After execution:

1. wrong_CONVEYOR_IOBUFFER_process_alarm *(2.group)*
2. wrong_part2_request *(3.group)*
3. wrong_part3_request *(3.group)*
4. iobuffer_entry_already_empty *(5.group)*

To implement this scenario, CONVEYOR_IOBUFFER process has been changed in the code of the robot. In the correct one, if this request comes, robot tries to find the first nearest same typed part, waiting for this request. However, in this scenario, it tries to find the first empty cup index. Also, when this request comes, the part that makes the request should be measured by CMM. This is also controlled by the event checker. The empty cup cannot have this information, so the 3.group violations occur. Type 1 is different from type 2 and 3, while they are manufactured only by one machines, type 1 is processed by both of them, so to reach the violation state about the route of type 1 is hard than the others.

2. Robot tries to take a part from CNC1 entry while CNC1 entry is already empty:

Example for the Violation of Requirement Groups:

1. Group 2 : Request-Process Consistency
2. Group 5 : Already empty hardware

After execution:

1. wrong_CONVEYOR_CNC1_process_alarm *(2.group)*
2. wrong_part2_request *(3.group)*
3. cnc1_entry_already_empty *(5.group)*

Same as the previous one, the process for CONVEYOR_CNC1 request of robot has been changed. While in the correct code, robot tries to find a part that matches the request, in this fault-injected simulation, it only finds the first empty cup. Also, event checker controls that from where the part comes, in this one, it must comes from IOBuffer, but in an empty cup, checker can not reach this information, so the route error raises.

3. Robot tries to take a part from CNC2 entry while CNC2 entry is already empty:

Example for the Violation of Requirement Groups:

1. Group 2 : Request-Process Consistency
2. Group 5 : Already empty hardware

After execution:

1. wrong_CONVEYOR_CNC2_process_alarm *(2.group)*
2. cnc2_entry_already_empty *(5.group)*
3. wrong_part3_request *(3.group)*

This entry is used by both CNC2 and CMM, so the change in the code is done for both requests, CONVEYOR_CNC2 and CONVEYOR_CMM. However, the run does not reach the state where the violation occurs for CONVEYOR_CMM request.

None of the runs has ended in this scenario, so the checker cannot control the violations that can be happen at the end of the simulation.

4.3.9 Scenario 9: Loading an already full cup

Robot cannot load a part onto the conveyor entry, which has already full. This scenario is split into three cases because there are three specific entries on the conveyor.

1. Robot tries to put a part into the IOBUFFER entry while IOBUFFER entry is already full:

Example for the Violation of Requirement Groups:

1. Group 2 : Request-Process Consistency
2. Group 4 : Limited Hardware Capacity

After execution:

1. wrong_IOBUFFER_process_alarm *(2.group)*
2. iobuffer_entry_capacity_exceeded (4.group)

To implement this scenario, in the code of the robot, the process for IOBUFFER request has been changed. In the correct one, if this request comes, robot tries to find the nearest free cup. However, in this scenario, it doe not try to find an empty cup, so while putting the part onto this entry, this entry may already have a part.

2. Robot tries to put a part into the CNC1 entry while the CNC1 entry is already full:

Example for the Violation of Requirement Groups:

1. Group 2 : Request-Process Consistency
2. Group 4 : Limited Hardware Capacity

After execution:

1. cnc1_entry_capacity_exceeded (4.group)
2. wrong_CNC1_process_alarm *(2.group)*

Same as the previous one, the process for CNC1 request of robot has been changed. While in the correct code, robot tries to find the first empty cup, in the fault injected one, it directly loads the part onto this entry.

3. Robot tries to put a part into the CNC2 entry while the CNC2 entry is already full:

Example for the Violation of Requirement Groups:

1. Group 2 : Request-Process Consistency
2. Group 4 : Limited Hardware Capacity

After execution:

1. cnc2_entry_capacity_exceeded (4.group)
2. wrong _CNC2_process_alarm *(2.group)*
3. wrong _CMM_process_alarm *(2.group)*

This entry is used by both CNC2 and CMM, so the change in the code is done for both requests, CNC2 and CMM. Robot directly puts the part onto this entry, without controlling whether it is full or empty.

None of the runs has ended because when the robot puts the part over another, the first part on the cup is erased and only the new part is seen by the system.

4.3.10 Scenario 10: Problem in conveyor rotation

Conveyor always rotates by exactly how many cups wanted. In this scenario, conveyor rotates by fourteen cups more. Therefore, when robot wants a free cup, or a cup with a part matching the request, the system finds the exact index of that entry, but the conveyor rotates one more tour to indexing that cup.

Example for the Violation of Requirement Groups:

1. Group 8 : Conveyor Rotation

After execution:

1. conveyor_rotate_problem *(8.group)*

For this one, only conveyor_rotate_problem alarm raises, and because of the extra turns, the incoming part count decreases approximately by half.

4.3.11 Scenario 11: Robot at wrong side

Robot has two position in the cell and it carries out half of the defined processes on the left side, and on the right side, it performs the others. Robot's start position is left side. In this scenario, robot can not move from its start position to the right side of the system.

Example for the Violation of Requirement Groups:

1. Group 9 : Robot sides

After execution:

1. robot_at_wrong_side *(9.group)*

In the validated simulation code according to the specified validation requirements, the side of the robot is not controls while the robot performs the request. Because this fault injected one is the new version with comments at robot movement code, everything seems ok, but not in the point of the checker.

4.3.12 Scenario 12: Problem in the direction of conveyor rotation

The conveyor rotates in the clockwise direction. In this scenario, the only change is that the conveyor turns in the counter-clockwise direction.

1. The system finds the part matching the currently processed request in the correct direction and evaluates by how many cups the conveyor must rotate, but the conveyor turns in the wrong direction:

Example for the Violation of Requirement Groups:

1. Group 2 : Request-Process Consistency

After execution:

1. wrong_IOBUFFER_process_alarm *(2.group)*
2. wrong_CNC1_process_alarm *(2.group)*
3. wrong_CNC2_process_alarm *(2.group)*
4. wrong_CMM_process_alarm *(2.group)*
5. wrong_CONVEYOR_CNC1_process_alarm *(2.group)*
6. wrong_CONVEYOR_CNC2_process_alarm *(2.group)*

The process of finding out the requested part is done in the clockwise direction, but the conveyor rotate in the counter clockwise direction, so the correct parts or free cups wanted cannot be indexed correctly. Two of the expected alarms do not raise because the system cannot reach at those violated states.

2. At this time, also the requested parts and the free cups are found out in the reverse direction on the conveyor, and conveyor rotates in this direction:

Example for the Violation of Requirement Groups:

1. No violation

After execution:

1. No violation

Since this state does not effect the correct run of the system, there is no violation. Only the entering part count of the system decreases because rotating in the reverse direction means rotating by more cups than in the normal turn.

4.3.13 Scenario 13: Not processing some parts

All three part types defined for this the system must be manufactured by at least one CNC. Afterwards, they are measured by CMM. In this scenario, some parts are measured without processed by any CNC.

Example for the Violation of Requirement Groups:

1. Group 3 : Part Routes Consistency
2. Group 6 : Visitation Consistency
3. Group 7 : Time Consistency

After execution:

1. wrong_part1_request *(3.group)*
2. wrong_part2_request *(3.group)*
3. wrong_part3_request *(3.group)*
4. cnc1_time_problem *(7.group)*
5. cnc2_time_problem *(7.group)*

6. CNC1_visitation_inconsistent *(6.group)*
7. CNC2_visitation_inconsistent *(6.group)*

All expected violations occur for this scenario. The part routes are not correct so the third group alarms raise. CNC1 and CNC2 are not used as evaluated according to the part types when a new part enters the system. Therefore, time problem and visitation inconsistency problem for their usage happen.

4.3.14 Scenario 14: Directly loading parts into CNC1 from IOBuffer

The parts cannot be transferred directly from one machine or input buffer to another machine or output buffer. In all transfers, conveyor must be used even directly putting the part has shorter way. In this scenario, the robot unloads the parts of the type 1 or 2 from input buffer and directly loads them into the CNC1.

Example for the Violation of Requirement Groups:

1. Group 2 : Request-Process Consistency
2. Group 3 : Part Routes Consistency

After execution:

1. wrong_part1_request *(3.group)*
2. wrong_part2_request *(3.group)*
3. wrong_IOBUFFER_process_alarm *(2.group)*
4. part_count_safe3 *(1.group)*
5. part_count_safe3_type1 *(1.group)*
6. part_count_safe3_type2 *(1.group)*
7. conveyor_capacity_exceeded (4.group)

At the end of the IOBUFFER request, the checker wants to see that the IOBuffer is full with the carried part. Because it is empty at the end of the process, wrong_IOBUFFER_process_alarm raises. If by wrongly, this entry is full (because the conveyor does not rotate to index a free cup to that entry), then onto_conveyor event is recognized by the event recognizer, then the count of current parts on the conveyor increases which causes the last two violations.

4.3.15 Scenario 15: Loading parts onto the wrong cups

There are three specific positions for the conveyor according to the machines, conveyor fixes its entries to these positions and robot can load or unload only the parts on the indexed entries. In this scenario, robot loads the part onto the just one front cup of the indexed one.

Example for the Violation of Requirement Groups:

1. Group 2 : Request-Process Consistency
2. Group 4 : Limited hardware capacity

After execution:

1. part_count_safe3 *(1.group)*
2. part_count_safe3_type1 *(1.group)*
3. part_count_safe3_type2 *(1.group)*
4. part_count_safe3_type3 *(1.group)*
5. cnc1_entry_capacity_exceeded (4.group)

The run gets stuck at the beginning, because the system indexes the first free cup to the IOBuffer entry, but the robot puts the part into the CNC1 entry. Therefore, the capacity of CNC1 entry exceeds. The other expected violations do not occur, since the system does not com into those violated states.

4.3.16 Scenario 16: Changing CNC places

CNC1 is on the left of the cell, and CNC2 is on the right. In this scenario, the places of these two machines are changed, CNC2 is on the left, CNC1 is on the right. Robot starts position is still left side.

Example for the Violation of Requirement Groups:

1. Group 3 : Part Routes Consistency

After execution:

1. wrong_part1_request *(3.group)*

The position of the two machines are changed, but the flow concept is not disturbed. Therefore, the only problem for this scenario is with the parts of type 1. In normal flow, these parts, manufactured in the left machine and then in the right machine. Now, they first processed by CNC2, and then manufactured by CNC1.

4.3.17 Scenario 17: Conveyor rotation by 1, 3 or 10 cups only

Conveyor can rotate by at least one cup and at most 13 cups. It may rotate by more than 13 cups, but this is not an acceptable situation. In this, scenario, it rotates only by 1 cup, 3 cups or 10 cups. When the part enters to the system, it is on the IOBuffer entry. When the conveyor rotates by one cup, it is at the CNC1 entry index; from this point, when the conveyor rotates by three cups, the parts is at the CNC2 entry index. Finally, after conveyor rotates by ten cups, the part is at the starting index position, which is the IOBuffer entry.

Example for the Violation of Requirement Groups:

1. No violation

After execution:

1. No violation

The simulation does not end, also the checker does not give any violation since the system cannot be in a state, which violates the validation rules. For this scenario, where the error is done cannot be found by monitoring MaC, because there is no specific validation rule has though for such a situation.

4.3.18 Scenario 18: Stop working after 7 hours

The working duration of the system is 7 hours. After this duration, system continues to run until the last part leaves the system. In this scenario, after 7 hours, system stops to run with a number of unprocessed parts.

Example for the Violation of Requirement Groups:

1. Group 1 : Preservation of Parts
2. Group 6 : Visitation Consistency
3. Group 7 : Time Consistency

After execution:

1. CNC1_visitation_inconsistent *(6.group)*
2. CNC2_visitation_inconsistent *(6.group)*
3. CMM_visitation_inconsistent *(6.group)*
4. cmm_time_problem *(7.group)*
5. part_out_count_error *(1.group)*
6. part_out_count_error1 (1.group)
7. part_out_count_error2 (1.group)
8. part_out_count_error3 (1.group)

All the expected violations, which raise at the end of the simulation run is caught by the checker. Machines are not used by evaluated times, so visitation inconsistency occurs. However, there is a problem with the usage time of cnc1 and cnc2. Although, there must be a time inconsistency for these machines because of the error bounds, it seems everything is ok. Also, the entering part count is not equal to the exiting part count at the end of the run.

4.3.19 Scenario 19: Processing the parts according to their IDs

The parts are processed according to their types. If there is a request, then the system finds the first matching part with this request on the conveyor. Also, the parts have identification numbers, which are not used in the correct simulation code. However, for this scenario, these identification numbers are used, so which ID makes the request, that part is processed.

Example for the Violation of Requirement Groups:

1. No violation

After execution:

1. No violation

There is no violation at the end of the simulation run, only the entering part count slightly decreases. This is because of that the rotate duration of the conveyor increases to index the cup with specific ID. While there may be a part matching with this request is too near, it must turn for that ID.

4.3.20 Scenario 20: Rerotation of incorrectly manufactured parts

All the parts leave the cell after manufactured, no matter whether they are processed correctly or not. The result of CMM is ignored, because the assumption is that, if a part is not manufactured correctly, it is put into the output buffer as the others and if the customer decides that it must be reprocessed, he can put it into the input buffer again. This is same as the current of METUCIM Lab for such a situation. In this scenario, the wrongly manufactured parts rerotate to be remanufactured in the cell.

Example for the Violation of Requirement Groups:

1. Group 3 : Part Routes Consistency
2. Group 6 : Visitation Consistency
3. Group 7 : Time Consistency

After execution:

1. CNC1_visitation_inconsistent *(6.group)*
2. CNC2_visitation_inconsistent *(6.group)*
3. CMM_visitation_inconsistent *(6.group)*
4. cnc1_time_problem *(7.group)*
5. cnc2_time_problem *(7.group)*
6. cmm_time_problem *(7.group)*
7. wrong_part1_request *(3.group)*
8. wrong_part2_request *(3.group)*
9. wrong_part3_request *(3.group)*

Because of the rerotation, the three machines, CNC1, CNC2 and CMM, are used more than evaluated. The reason is that the evaluation is done only when the parts are put on the conveyor from the input buffer, which means that a new part enters to the system. Also, all parts must leave the system after measured by CMM. In this case, they do not leave, so they violate their process routes.

4.3.21 Scenario 21: Robot can not take parts

The assumption is that, since the duration is 7 hours, which is a short time, there is nothing wrong for any run. In this scenario, the robot cannot take the parts. Parts stay in machines or on the input buffer or on the conveyor cups. To avoid to stuck the run at some point, these are handled one by one.

1. Robot can not take part from CNC1:

Example for the Violation of Requirement Groups:

1. Group 2 : Request-Process Consistency
2. Group 5 : Already empty hardware

After execution:

1. part_count_safe3 *(1.group)*
2. part_count_safe3_type1 *(1.group)*
3. part_count_safe3_type2 *(1.group)*
4. part_count_safe3_type3 *(1.group)*
5. wrong_CNC1_process_alarm *(2.group)*
6. wrong_CONVEYOR_CMM_process_alarm *(2.group)*
7. cnc2_entry_already_empty *(5.group)*
8. robot_process_error *(5.group)*

Robot cannot take the part, but it goes on doing its job without knowing this, so the CNC1 entry behaves like that there is a part on it. This causes the other following violations while the simulation run is continuing.

2. Robot can not take part from CNC2:

Example for the Violation of Requirement Groups:

1. Group 2 : Request-Process Consistency
2. Group 5 : Already empty hardware

After execution:

1. part_count_safe3 *(1.group)*
2. part_count_safe3_type1 *(1.group)*
3. part_count_safe3_type2 *(1.group)*
4. part_count_safe3_type3 *(1.group)*
5. wrong_CNC2_process_alarm *(2.group)*
6. wrong_CMM_process_alarm *(2.group)*

7. robot_process_error *(5.group)*

The reason of these violations is same as the previous one. Also, the robot tries to put a part while there is no part at the mount of itself, which causes robot_process_error alarm to be raised like the above scenario again.

3. Robot can not take part from CMM:

Example for the Violation of Requirement Groups:

1. Group 2 : Request-Process Consistency
2. Group 5 : Already empty hardware

After execution:

1. part_count_safe3 *(1.group)*
2. part_count_safe3_type1 *(1.group)*
3. part_count_safe3_type2 *(1.group)*
4. part_count_safe3_type3 *(1.group)*
5. cnc2_entry_capacity_exceeded *(4.group)*
6. wrong _CNC2_process_alarm *(2.group)*
7. robot_process_error *(5.group)*

Since the robot uses the same entry for CNC2 and CMM requests, although CNC2 is left behind, the process error of this raises. The other reasons mentioned above for the previous two scenarios are still valid for this one.

4. Robot can not take part from the conveyor:

Example for the Violation of Requirement Groups:

1. Group 2 : Request-Process Consistency
2. Group 5 : Already empty hardware

After execution:

1. part_count_safe3 *(1.group)*
2. part_count_safe3_type1 *(1.group)*
3. part_count_safe3_type2 *(1.group)*
4. part_count_safe3_type3 *(1.group)*
5. wrong_CONVEYOR_CNC1_process_alarm *(2.group)*
6. wrong_CONVEYOR_CNC2_process_alarm *(2.group)*
7. robot_process_error *(5.group)*

This scenario means that no parts can be processed by the machines, all parts entering the system stay on the conveyor, so they cannot reach even to CMM.

5. Robot can not take part from the input buffer:

Example for the Violation of Requirement Groups:

1. Group 2 : Request-Process Consistency
2. Group 5 : Already empty hardware

After execution:

1. part_count_safe3 *(1.group)*
2. part_count_safe3_type1 *(1.group)*
3. part_count_safe3_type2 *(1.group)*
4. part_count_safe3_type3 *(1.group)*
5. wrong_IOBUFFER_process_alarm *(2.group)*
6. robot_process_error *(5.group)*

No part can enter the system, but the event recognizer sees the onto_conveyor event, and it increases the count of the entering parts to the system. This causes the first violation.

4.3.22 Scenario 22: Not starting in a cold start situation

When the cell begins to run, it must be in a cold start situation, which means that there must not be any part at the cell, all hardware must be empty. In this scenario, however, it starts while there are several parts on the conveyor.

Example for the Violation of Requirement Groups:

1. Group 2 : Request-Process Consistency
2. Group 4 : Limited hardware capacity

After execution:

1. wrong_IOBUFFER_process_alarm *(2.group)*
2. iobuffer_entry_capacity_exceeded (4.group)

The system does not know the type of the parts, which are on the cell when the run starts. The system can know the part types, only when it places that part onto any hardware. Therefore, while processing the requests, it behaves like that there is no part on that cup. This is because the first simulation code has been written according to the correct situations, and this scenario is the rewritten version of that one with fault.

4.3.23 Scenario 23: Entering more than 14 parts

The capacity of the system, while it is running, is fourteen, this means that there can be at most fourteen parts in the cell at any time. In this scenario, the cell accepts new parts without controlling the current part count.

Example for the Violation of Requirement Groups:

1. Group 1 : Preservation of Parts

After execution:

1. capacity_exceeded *(1.group)*

The simulation run does not end, because exceeding capacity means, free cups can be found to put processed parts by the machines. Therefore, a deadlock occurs.

4.3.24 Scenario 24: Problem with two consecutive exit events

For validation rule group 10, this scenario is written. After the robot puts the part onto the output buffer, it tries to process the same job again. System behaves as the part is still on the conveyor to reach such a violated state located in group 10.

Example for the Violation of Requirement Groups:

1. Group 10 : Minimum time difference between two consecutive exit events

After execution:

1. min_time_difference *(10.group)*
2. wrong_CONVEYOR_IOBUFFER_process_alarm *(2.group)*
3. wrong_part2_request *(3.group)*
4. wrong_part3_request *(3.group)*
5. iobuffer_entry_already_empty *(5.group)*

As expected, the min_time_difference violation occurs. This scenario is an example of that, in the system, the minimum duration between any two jobs can be checked. The superfluous alarms raises because the system is forced to reach that state. Normally, if the entry is empty, it does not do anything, but in this situation, robot still tries to get a part from an empty cup, which is emptied by itself, a little ago.

4.3.25 Scenario 25: New parts entered by the customer

The parts are normally loaded onto the IOBuffer entry from the input buffer by robot. In this scenario, it is assumed that the customer does this job instead of the robot.

Example for the Violation of Requirement Groups:

1. Group 1 : Preservation of Parts
2. Group 2 : Request-Process Consistency

After execution:

1. wrong_IOBUFFER_process_alarm *(2.group)*
2. part_count_safe1 *(1.group)*
3. part_count_safe1_type1 *(1.group)*
4. part_count_safe1_type2 *(1.group)*
5. part_count_safe1_type3 *(1.group)*
6. part_count_safe3 *(1.group)*
7. part_count_safe1_type1 *(1.group)*
8. part_count_safe1_type2 *(1.group)*
9. part_count_safe1_type3 *(1.group)*

Event recognizer can not see a new part entering the system event because this event is defined as a robot dependent event. Therefore, the exiting part count of the system is always greater than the entering part count, which causes the violation.

4.3.26 Scenario 26: Conveyor does not rotate

In this scenario, the “rotate” action of the conveyor is forgotten in the implementation, but this is not noticed.

Example for the Violation of Requirement Groups:

1. Group 4 : Limited hardware capacity

2. Group 5 : Already empty hardware

After execution:

1. wrong_IOBUFFER_process_alarm *(2.group)*
2. wrong_CONVEYOR_CMM_process_alarm *(2.group)*
3. wrong_CONVEYOR_CNC1_process_alarm *(2.group)*
4. wrong_CONVEYOR_IOBUFFER_process_alarm *(2.group)*
5. wrong_CONVEYOR_CNC2_process_alarm *(2.group)*
6. iobuffer_entry_capacity_exceeded *(4.group)*
7. wrong_part1_request *(3.group)*
8. wrong_part2_request *(3.group)*
9. wrong_part3_request *(3.group)*
10. cnc1 _entry_already_empty *(5.group)*
11. cnc2_entry_already_empty *(5.group)*

"The conveyor does not rotate" means that the parts are put onto the cup at the iobuffer entry position, so the capacity of this cups exceeds. Also, the program continues to run as if there is no such problem, so the robot tries to get a part from the empty hardware of the cell.

4.3.27 Scenario 27: Changing the type of a part

There must be part consistency according to types as the type of a part does not change. In this scenario, the type of some part is 2 and, it becomes type 1 at some instant of its processing.

Example for the Violation of Requirement Groups:

1. Group 1 : Preservation of Parts

After execution:

1. part_count_safe3_type1 *(1.group)*
2. part_count_safe3_type2 *(1.group)*
3. part_out_count_error1 *(1.group)*

4. part_out_count_error2 *(1.group)*
5. part_count_safe1_type1 *(1.group)*
6. CNC2_visitation_inconsistent *(6.group)*
7. cnc2_time_problem *(7.group)*

In this scenario, the preservation of the parts according to their types are checked. There is no change in the count of the parts in the cell. However, the type of any part is changed. This situation can not be caught by the preservation of parts according to only their counts. Therefore, the same preservation events, conditions, alarms and safety properties are defined for each part type.

CHAPTER 5

CONCLUSION AND DISCUSSION

In this work, we studied the assertion checking method, and tried to use it as a validation technique. Finding an example from the real world should be effective to demonstrate the results of the application of assertion checking method for validation. Therefore, we have modeled the METUCIM Laboratory, which is a single manufacturing cell. The model is process oriented, and is implemented by using Java thread library. The simplicity of this example chosen made our work have been understood clearly. Also, we found luckily, MaC tool, which serves our aim. At the beginning, it seemed that MaC would make our life easier, because it exactly fit our work; however, the problems with MaC made this work much slower. Finally, the simulation was finished, and the working version of the MaC scripts including all validation requirements were written. The last missing thing was the demonstration of correctness of the simulation runs with respect to the validation requirements of the real world. For this, we created a lot of scenarios. Each scenario was simulated and their runs were monitored and checked by using the MaC scripts.

It should be emphasized that we do not try to validate the simulation model, we only validate the current run of the simulation against a set of validation requirements. This is because of that we use the assertion checking method to achieve this, which is one of the dynamic V&V techniques, and in dynamic techniques run-time monitoring is required. One of the limitations of run-time monitoring is that it is virtually impossible to monitor all possible runs of a model. Thus, it is practical to validate a simulation model itself in this way.

In this thesis, one person has done all the steps, analyzed the system, extracted the validation requirements, made the design, wrote the simulation code, and the MaC scripts. If all jobs are done by one person, the mistake or misunderstanding is repeated at all steps of the work. One person can make a wrong analyses, and by this wrong result, he/she builds the wrong system and writes wrong validation scripts. As a result, nothing can be caught, everything looks like working properly. It seems a chaos at first glance, but if we think of a project team with at least two people, this work gets much more meaning. If one makes a mistake or misunderstands something, at the other side of the project (the simulation code side or monitoring and checking side), it can be realized that either the simulation model or code is not right, or there is something wrong with the defined validation requirements. In fact, model development and derivation of assertions are independent processes, best performed by different persons.

In this work, we tried to show that assertion checking can be used as a validation method. This approach can be effective for run-time validations of discrete event simulation model in general. Of course, the validation requirements and necessary tools must be tailored for each specific application. For this aim, we used a run-time monitoring and checking tool, MaC version 0.99. Because, we encountered some limitations and problems with MaC, to overcome them, the validation requirements were defined in a limited way, such as defining three same classes for the three entries of the conveyor. For the advanced versions of the MaC tool, this work can be more improved. This is a core study, and it shows that with a more mature tool, more complex simulations and requirements can be handled easily.

Consequently, as the results of the test studies show that, the aim gains success, except from the *time consistency* of machines. This is because of using the system clock and sleep method of threads. This problem can be handled in a mature tool or again with MaC, seeing this problem, instead of making the threads sleep, other ways can be used. However, this missing was tried to close by another assertion validation, *visitation consistency*. In generally, the results are successful and show that, by using assertion checking, no mistake can be escaped from the system. Although there can be superfluous violations, the important thing here is that if there is a violation the system warns the user, since it is not intended to provide diagnostics for locating errors. Also, although the case study is based on an actual system (METUCIM Lab), the method is applicable to hypothetical systems as well. Because, the validation is performed with respect to a conceptual model.

REFERENCES

1. Sargent, R. G., Simulation Model Verification and Validation, *Proc. of the 2000 Winter Simulation Conf.*, 2000.

2. Balci, O., Verification, Validation, and Accreditation, *Proc. of the 1998 Winter Simulation Conf.*, 1998.

3. Preece, A., Evaluating Verification and Validation Methods in Knowledge Engineering.

4. Whitner, R.B., Balci, O., "Guidelines for selecting and using simulation model verification techniques" in E.A. MacNair, K.J. Musselman, & P.Heidelberger (Eds.), Proceeding of the 1989 Winter Simulation Conference, pp. 559-568, IEEE, Piscataway, NJ, 1989.

5. Defense Modeling and Simulation Office, DoD VV&A Recommended Practices Guide (RPG Build 2) "V&V Techniques", 15 August 2001.

6. Harmon, S.Y., Gross, D. C., Youngblood, S. M., "Why Validation," 1999 Summer Computer Simulation Conference (SCSC) Proceedings, Chicago, IL, July 1999.

7. Defense Modeling and Simulation Office, DoD VV&A Recommended Practices Guide (RPG Special Topic) "Validation", 30 November 2000.

8. Ronsse, M., Maebe, J., Bosschere, K.D., Software Instrumentation Using Dynamic Techniques.

9. DeMillo, R.A., McCracken, W.M., Martin, R.J., Passafiume, J.F., Software Testing and Evaluation, 1987.

10. Monitoring and Checking Framework, http://www.cis.upenn.edu/~rtg/mac/

11. Kim, M., Kannan, S., Lee, I., Sokolsky, O., Java-MaC: a Run-time Assurance Tool for Java Programs, 2001, available from URL in [10].

12. Languages in the MaC prototype implementation, MaC Research, University of Pennsylvania, , available from URL in [10].

13. Pinedo, M., "Scheduling, theory, algorithms, and systems", Prentice Hall, p. 25, 1995.

14. Candar, T., Development of an Agent Based Flexible Manufacturing Cell Controller Using Distributed Internet Applications, MS Thesis, Middle East Technical University, Ankara, 2000.

15. Luggen, W. W., Flexible Manufacturing Cells and Systems, Prentice Hall, p. 19, 378, year 1991.

16. Encyclopedia of Production and Manufacturing Management, http://reference.kluweronline.com.

17. http://manta.cs.vt.edu/cs6204/Assignments/Presentations/Mirza.ppt

APPENDIX A

PEDL SCRIPT OF METUCIM LAB

```
MonScr METUCIM

        //exported events for defining high-level events and conditions more easily

        export event startPgm,
                robotFull_event, robotEmpty_event,
                robotFull_event1, robotEmpty_event1,
                robotFull_event2, robotEmpty_event2,
                robotFull_event3, robotEmpty_event3,
                cnc1Full_event, cnc1Empty_event,
                cnc1Full_event1, cnc1Empty_event1,
                cnc1Full_event2, cnc1Empty_event2,
                cnc1Full_event3, cnc1Empty_event3,
                cnc2Full_event, cnc2Empty_event,
                cnc2Full_event1, cnc2Empty_event1,
                cnc2Full_event2, cnc2Empty_event2,
                cnc2Full_event3, cnc2Empty_event3,
                cmmFull_event, cmmEmpty_event,
                cmmFull_event1, cmmEmpty_event1,
                cmmFull_event2, cmmEmpty_event2,
                cmmFull_event3, cmmEmpty_event3,
                cnc1_start_work, cnc2_start_work, cmm_start_work,
                cnc1_stop_work, cnc2_stop_work, cmm_stop_work,
                IOBufferEntryFull_event, IOBufferEntryEmpty_event,
                IOBufferEntryFull_event1, IOBufferEntryEmpty_event1,
                IOBufferEntryFull_event2, IOBufferEntryEmpty_event2,
                IOBufferEntryFull_event3, IOBufferEntryEmpty_event3,
                cnc1EntryFull_event, cnc1EntryEmpty_event,
                cnc1EntryFull_event1, cnc1EntryEmpty_event1,
                cnc1EntryFull_event2, cnc1EntryEmpty_event2,
                cnc1EntryFull_event3, cnc1EntryEmpty_event3,
                cnc2EntryFull_event, cnc2EntryEmpty_event,
                cnc2EntryFull_event1, cnc2EntryEmpty_event1,
                cnc2EntryFull_event2, cnc2EntryEmpty_event2,
                cnc2EntryFull_event3, cnc2EntryEmpty_event3,
                from_conveyor, onto_conveyor, from_conveyor1, onto_conveyor1,
                from_conveyor2, onto_conveyor2, from_conveyor3, onto_conveyor3,
                new_part, exit_part , exit_part1 , exit_part2 , exit_part3 ,
                control_start, control_end, abnormal_rotate,
                part1_path_event, part2_path_event, part3_path_event,
                CNC1_will_used, CNC2_will_used, CMM_will_used,
                CNC1_has_used, CNC2_has_used, CMM_has_used,
                wrong_IOBUFFER_process, wrong_CNC1_process,
                wrong_CNC2_process, wrong_CMM_process,
                wrong_CONVEYOR_IOBUFFER_process,
                wrong_CONVEYOR_CNC1_process,
                wrong_CONVEYOR_CNC2_process,
                wrong_CONVEYOR_CMM_process,
                new_part_type1, new_part_type2, new_part_type3,
                wrong_robot_place,endPgm;

        //Monitored attributes of the classes
        monobj int METUCIM.val_part_cnt_control;
        monobj int METUCIM.mac1;
        monobj boolean METUCIM.end_threads;
        monobj int Robot.val_full;
        monobj int Cnc1.val_full;
        monobj int Cnc2.val_full;
        monobj int Cmm.val_full;
        monobj int Cnc1.part_type;
        monobj int Cnc2.part_type;
        monobj int Cmm.part_type;
        monobj int Cnc1.start_work;
        monobj int Cnc2.start_work;
        monobj int Cmm.start_work;
        monobj int IOBUFFER_Cup_Entry.val_full;
        monobj int IOBUFFER_Cup_Entry.part_type;
        monobj int CNC1_Cup_Entry.val_full;
        monobj int CNC1_Cup_Entry.part_type;
        monobj int CNC2_Cup_Entry.val_full;
        monobj int CNC2_Cup_Entry.part_type;
        monobj int Robot.x_request_type;
        monobj int Robot.x_part_type;
        monobj int Robot.x_from;
        monobj int Robot.after_turn;
        monobj int Robot.robot_place;
        monobj int Conveyor.turn_number;
        monobj int Conveyor.turn_flag;
        monobj int IOBuffer.first_element;

        //Also, the main method is monitored
        monmeth void METUCIM.main(String[]);

        //Current request in the cell
        condition IOBUFFER                  = (Robot.x_request_type==1);
        condition CNC1                      = (Robot.x_request_type==2);
        condition CONVEYOR_IOBUFFER = (Robot.x_request_type==3);
        condition CONVEYOR_CNC1         = (Robot.x_request_type==4);
        condition CONVEYOR_CNC2         = (Robot.x_request_type==5);
        condition CONVEYOR_CMM          = (Robot.x_request_type==6);
        condition CNC2                    = (Robot.x_request_type==7);
        condition CMM                     = (Robot.x_request_type==8);

        //which component of the system makes that request,
        //this means that the part lastly in that component
        condition from_IOBUFFER        = (Robot.x_from==1);
        condition from_CNC1            = (Robot.x_from==2);
        condition from_CNC2            = (Robot.x_from==7);
        condition from_CMM             = (Robot.x_from==8);

        //while robot puts a part, or gets a part,
        //to control that to which component the part belongs
        condition control_ok = (METUCIM.val_part_cnt_control == 1);
        condition end_control = (METUCIM.val_part_cnt_control == 0);

        event control_start = start(control_ok);
        event control_end = start(end_control);

        //start of the simulation
        event startPgm = StartM(METUCIM.main(String[]));

        //simulation ends
        event endPgm = start(METUCIM.end_threads==true);

        //a new part is loaded into the cell
        event new_part = start(IOBUFFER && robotFull_cond);

        //what is the type of this new part?
        event new_part_type1 = start(IOBUFFER && robotFull_cond &&
                                Robot.x_part_type==1);
        event new_part_type2 = start(IOBUFFER && robotFull_cond &&
                                Robot.x_part_type==2);
        event new_part_type3 = start(IOBUFFER && robotFull_cond &&
                                Robot.x_part_type==3);

        //part exit from the cell
        event exit_part = start(CONVEYOR_IOBUFFER && (Robot.after_turn==3) &&
                                robotEmpty_cond && IOBufferEntryEmpty_cond);
        event exit_part1 = start(CONVEYOR_IOBUFFER && (Robot.after_turn==3) &&
                                robotEmpty_cond && IOBufferEntryEmpty_cond)
                        when Robot.x_part_type==1;
        event exit_part2 = start(CONVEYOR_IOBUFFER && (Robot.after_turn==3) &&
                                robotEmpty_cond && IOBufferEntryEmpty_cond)
                        when Robot.x_part_type==2;
        event exit_part3 = start(CONVEYOR_IOBUFFER && (Robot.after_turn==3) &&
                                robotEmpty_cond && IOBufferEntryEmpty_cond)
                        when Robot.x_part_type==3;

        //the below components become full or empty
        event robotFull_event = start(robotFull_cond);
        event robotFull_event1 = start(robotFull_cond) when Robot.x_part_type==1;
        event robotFull_event2 = start(robotFull_cond) when Robot.x_part_type==2;
        event robotFull_event3 = start(robotFull_cond) when Robot.x_part_type==3;
        event robotEmpty_event = start(robotEmpty_cond);
        event robotEmpty_event1 = start(robotEmpty_cond) when Robot.x_part_type==1;
        event robotEmpty_event2 = start(robotEmpty_cond) when Robot.x_part_type==2;
        event robotEmpty_event3 = start(robotEmpty_cond) when Robot.x_part_type==3

        event cnc1Full_event = start(cnc1Full_cond);
        event cnc1Full_event1 = start(cnc1Full_cond) when Robot.x_part_type==1;
        event cnc1Full_event2 = start(cnc1Full_cond) when Robot.x_part_type==2;
        event cnc1Full_event3 = start(cnc1Full_cond) when Robot.x_part_type==3

        event cnc1Empty_event = start(cnc1Empty_cond);
        event cnc1Empty_event1 = start(cnc1Empty_cond) when Robot.x_part_type==1;
        event cnc1Empty_event2 = start(cnc1Empty_cond) when Robot.x_part_type==2;
        event cnc1Empty_event3 = start(cnc1Empty_cond) when Robot.x_part_type==3

        event cnc2Full_event = start(cnc2Full_cond);
        event cnc2Full_event1 = start(cnc2Full_cond) when Robot.x_part_type==1;
        event cnc2Full_event2 = start(cnc2Full_cond) when Robot.x_part_type==2;
        event cnc2Full_event3 = start(cnc2Full_cond) when Robot.x_part_type==3

        event cnc2Empty_event = start(cnc2Empty_cond);
        event cnc2Empty_event1 = start(cnc2Empty_cond) when Robot.x_part_type==1;
        event cnc2Empty_event2 = start(cnc2Empty_cond) when Robot.x_part_type==2;
        event cnc2Empty_event3 = start(cnc2Empty_cond) when Robot.x_part_type==3

        event cmmFull_event = start(cmmFull_cond);
        event cmmFull_event1 = start(cmmFull_cond) when Robot.x_part_type==1;
        event cmmFull_event2 = start(cmmFull_cond) when Robot.x_part_type==2;
        event cmmFull_event3 = start(cmmFull_cond) when Robot.x_part_type==3

        event cmmEmpty_event = start(cmmEmpty_cond);
        event cmmEmpty_event1 = start(cmmEmpty_cond) when Robot.x_part_type==1;
        event cmmEmpty_event2 = start(cmmEmpty_cond) when Robot.x_part_type==2;
        event cmmEmpty_event3 = start(cmmEmpty_cond) when Robot.x_part_type==3

        event IOBufferEntryFull_event = start(IOBufferEntryFull_cond);
        event IOBufferEntryFull_event1 = start(IOBufferEntryFull_cond)
                                         when Robot.x_part_type==1;
        event IOBufferEntryFull_event2=start(IOBufferEntryFull_cond)
                                         when Robot.x_part_type==2;
        event IOBufferEntryFull_event3=start(IOBufferEntryFull_cond)
                                         when Robot.x_part_type==3

        event IOBufferEntryEmpty_event = start(IOBufferEntryEmpty_cond);
        event IOBufferEntryEmpty_event1 = start(IOBufferEntryEmpty_cond)
                                                  when Robot.x_part_type==1;
        event IOBufferEntryEmpty_event2 = start(IOBufferEntryEmpty_cond)
                                                  when Robot.x_part_type==2;
        event IOBufferEntryEmpty_event3 = start(IOBufferEntryEmpty_cond)
                                                  when Robot.x_part_type==3
        event cnc1EntryFull_event = start(cnc1EntryFull_cond);
        event cnc1EntryFull_event1 = start(cnc1EntryFull_cond)
                                         when Robot.x_part_type==1;
```

```
event cnc1EntryFull_event2 = start(cnc1EntryFull_cond)
                                        when Robot.x_part_type==2;
event cnc1EntryFull_event3 = start(cnc1EntryFull_cond)
                                        when Robot.x_part_type==3;

event cnc1EntryEmpty_event = start(cnc1EntryEmpty_cond);
event cnc1EntryEmpty_event1 = start(cnc1EntryEmpty_cond)
                                        when Robot.x_part_type==1;
event cnc1EntryEmpty_event2 = start(cnc1EntryEmpty_cond)
                                        when Robot.x_part_type==2;
event cnc1EntryEmpty_event3 = start(cnc1EntryEmpty_cond)
                                        when Robot.x_part_type==3;

event cnc2EntryFull_event = start(cnc2EntryFull_cond);
event cnc2EntryFull_event1 = start(cnc2EntryFull_cond)
                                        when Robot.x_part_type==1;
event cnc2EntryFull_event2 = start(cnc2EntryFull_cond)
                                        when Robot.x_part_type==2;
event cnc2EntryFull_event3 = start(cnc2EntryFull_cond)
                                        when Robot.x_part_type==3;

event cnc2EntryEmpty_event = start(cnc2EntryEmpty_cond);
event cnc2EntryEmpty_event1 = start(cnc2EntryEmpty_cond)
                                        when Robot.x_part_type==1;
event cnc2EntryEmpty_event2 = start(cnc2EntryEmpty_cond)
                                        when Robot.x_part_type==2;
event cnc2EntryEmpty_event3 = start(cnc2EntryEmpty_cond)
                                        when Robot.x_part_type==3;

//the below components start and finish operation,
event cnc1_start_work = start(Cnc1.start_work==1);
event cnc1_stop_work = end(Cnc1.start_work==1);
event cnc2_start_work = start(Cnc2.start_work==1);
event cnc2_stop_work = end(Cnc2.start_work==1);
event cmm_start_work = start(Cmm.start_work==1);
event cmm_stop_work = end(Cmm.start_work==1);

//robot can move from one side to the other in the cell
//it must be at the right side while operates a request
event wrong_robot_place = start((Robot.robot_place==1 &&
                                        Robot.x_request_type > 4 && robotFull_cond) ||
                                   (Robot.robot_place==2 &&
                                        Robot.x_request_type < 5 && robotFull_cond));

//conveyor must rotate by at least 1, at most 13 cups
event abnormal_rotate = start(Conveyor.turn_flag==1 &&
                         ((Conveyor.turn_number >0 && Conveyor.turn_number > 13) ||
                          (Conveyor.turn_number <0 && Conveyor.turn_number+14 <=0) ||
                           Conveyor.turn_number==0));

//the below components are full or empty
condition robotFull_cond = (Robot.val_full==1);
condition robotEmpty_cond = (Robot.val_full==0);
condition cnc1Full_cond = (Cnc1.val_full==1);
condition cnc1Empty_cond = (Cnc1.val_full==0);

condition cnc2Full_cond = (Cnc2.val_full==1);
condition cnc2Empty_cond = (Cnc2.val_full==0);

condition cmmFull_cond = (Cmm.val_full==1);
condition cmmEmpty_cond = (Cmm.val_full==0);

condition IOBufferEntryFull_cond = (IOBUFFER_Cup_Entry.val_full==1);
condition IOBufferEntryEmpty_cond = (IOBUFFER_Cup_Entry.val_full==0);

condition cnc1EntryFull_cond = (CNC1_Cup_Entry.val_full==1);
condition cnc1EntryEmpty_cond = (CNC1_Cup_Entry.val_full==0);

condition cnc2EntryFull_cond = (CNC2_Cup_Entry.val_full==1);
condition cnc2EntryEmpty_cond = (CNC2_Cup_Entry.val_full==0);

//control if there is something wrong with the routes of the parts
condition part1_path = !control_ok && !(Robot.x_part_type!=1 ||
                    (Robot.x_part_type==1 &&  (IOBUFFER ||
                       (CONVEYOR_CNC1 && from_IOBUFFER) || CNC1 ||
                       (CONVEYOR_CNC2 && from_CNC1) || CNC2 ||
                       (CONVEYOR_CMM  && from_CNC2) || CMM ||
                       (CONVEYOR_IOBUFFER && from_CMM))));

condition part2_path = !control_ok && !(Robot.x_part_type!=2 ||
                    (Robot.x_part_type==2 &&  (IOBUFFER ||
                       (CONVEYOR_CNC1 && from_IOBUFFER) || CNC1 ||
                       (CONVEYOR_CMM  && from_CNC1) || CMM ||
                       (CONVEYOR_IOBUFFER && from_CMM)) &&
                       !CONVEYOR_CNC2 && !CNC2));

condition part3_path = !control_ok && !(Robot.x_part_type!=3 ||
                    (Robot.x_part_type==3 &&  (IOBUFFER ||
                       (CONVEYOR_CNC2 && from_IOBUFFER) || CNC2 ||
                       (CONVEYOR_CMM  && from_CNC2) || CMM ||
                       (CONVEYOR_IOBUFFER && from_CMM)) &&
                       !CONVEYOR_CNC1 && !CNC1));

event part1_path_event = start(part1_path);
event part2_path_event = start(part2_path);
event part3_path_event = start(part3_path);

//the useage of the machines,
//how many times they will be used, and
//how many times they have been used.
event CNC1_will_used = new_part && start(Robot.x_part_type!=3);
event CNC2_will_used = new_part && start(Robot.x_part_type!=2);
event CMM_will_used = new_part;
event CNC1_has_used = end(Cnc1.val_full==1);
event CNC2_has_used = end(Cnc2.val_full==1);
event CMM_has_used = end(Cmm.val_full==1);
```

```
//control whether something wrong with processing of the requests
event wrong_IOBUFFER_process = start(control_ok && IOBUFFER &&
        !((Robot.after_turn==0) ||
        ((Robot.after_turn==1) &&
                robotEmpty_cond && IOBufferEntryEmpty_cond &&
                (IOBuffer.first_element==Robot.x_part_type)) ||
        ((Robot.after_turn==2) &&
                robotFull_cond && IOBufferEntryEmpty) ||
        ((Robot.after_turn==3) &&
                robotEmpty_cond && IOBufferEntryFull_cond &&
                (IOBUFFER_Cup_Entry.part_type==Robot.x_part_type))));

event wrong_CNC1_process = start(control_ok && CNC1 &&
        !((Robot.after_turn==0 && cnc1Full_cond) ||
         ((Robot.after_turn==1) &&
                robotEmpty_cond && cnc1EntryEmpty_cond &&
                cnc1Full_cond && (Cnc1.part_type==Robot.x_part_type)) ||
         ((Robot.after_turn==2) &&
                robotFull_cond && cnc1EntryEmpty_cond &&
                cnc1Empty_cond) ||
         ((Robot.after_turn==3) && robotEmpty_cond &&
                cnc1EntryFull_cond && cnc1Empty_cond &&
                (CNC1_Cup_Entry.part_type==Robot.x_part_type))));

event wrong_CNC2_process = start(control_ok && CNC2 &&
        !((Robot.after_turn==0 && cnc2Full_cond) ||
         ((Robot.after_turn==1)
                && robotEmpty_cond && cnc2EntryEmpty_cond &&
                cnc2Full_cond && (Cnc2.part_type==Robot.x_part_type)) ||
         ((Robot.after_turn==2) &&
                robotFull_cond && cnc2EntryEmpty_cond &&
                cnc2Empty_cond) ||
         ((Robot.after_turn==3) && robotEmpty_cond &&
                cnc2EntryFull_cond && cnc2Empty_cond &&
                (CNC2_Cup_Entry.part_type==Robot.x_part_type))));

event wrong_CMM_process = start(control_ok && CMM &&
        !((Robot.after_turn==0 && cmmFull_cond) ||
         ((Robot.after_turn==1) &&
                robotEmpty_cond && cnc2EntryEmpty_cond &&
                cmmFull_cond && (Cmm.part_type==Robot.x_part_type)) ||
         ((Robot.after_turn==2) &&
                robotFull_cond && cnc2EntryEmpty_cond &&
                cmmEmpty_cond) ||
         ((Robot.after_turn==3) &&
                robotEmpty_cond && cnc2EntryFull_cond &&
                cmmEmpty_cond &&
                (CNC2_Cup_Entry.part_type==Robot.x_part_type))));

event wrong_CONVEYOR_IOBUFFER_process =
        start(control_ok && CONVEYOR_IOBUFFER &&
        !((Robot.after_turn==0) ||
         ((Robot.after_turn==1) &&
                 robotEmpty_cond && iobufferEntryFull_cond &&
                (IOBUFFER_Cup_Entry.part_type==Robot.x_part_type)) ||
         ((Robot.after_turn==2) &&
                robotFull_cond && iobufferEntryEmpty_cond) ||
        ((Robot.after_turn==3) &&
                robotEmpty_cond && iobufferEntryEmpty_cond)));

event wrong_CONVEYOR_CNC1_process =
        start(control_ok && CONVEYOR_CNC1 &&
        !((Robot.after_turn==0) ||
         ((Robot.after_turn==1) &&
                robotEmpty_cond && cnc1EntryFull_cond &&
                cnc1Empty_cond &&
                 (CNC1_Cup_Entry.part_type==Robot.x_part_type)) ||
         ((Robot.after_turn==2) &&
                 robotFull_cond && cnc1EntryEmpty_cond &&
                 cnc1Empty_cond) ||
         ((Robot.after_turn==3) &&
                robotEmpty_cond && cnc1EntryEmpty_cond &&
                cnc1Full_cond &&
                (Cnc1.part_type==Robot.x_part_type))));

event wrong_CONVEYOR_CNC2_process =
        start(control_ok && CONVEYOR_CNC2 &&
         !((Robot.after_turn==0) ||
          ((Robot.after_turn==1) &&
                robotEmpty_cond && cnc2EntryFull_cond && cnc2Empty_cond
                && (CNC2_Cup_Entry.part_type==Robot.x_part_type)) ||
         ((Robot.after_turn==2) &&
                robotFull_cond && cnc2EntryEmpty_cond &&
                cnc2Empty_cond) ||
          ((Robot.after_turn==3) &&
                robotEmpty_cond && cnc2EntryEmpty_cond &&
                cnc2Full_cond && (Cnc2.part_type==Robot.x_part_type))));

event wrong_CONVEYOR_CMM_process =
        start(control_ok && CONVEYOR_CMM &&
        !((Robot.after_turn==0) ||
        ((Robot.after_turn==1) &&
                robotEmpty_cond && cnc2EntryFull_cond &&
                cmmEmpty_cond &&
                (CNC2_Cup_Entry.part_type==Robot.x_part_type)) ||
        ((Robot.after_turn==2) &&
                robotFull_cond && cnc2EntryEmpty_cond &&
                cmmEmpty_cond) ||
        ((Robot.after_turn==3) &&
                robotEmpty_cond && cnc2EntryEmpty_cond &&
                cmmFull_cond && (Cmm.part_type==Robot.x_part_type))));

//a part has been taken from the conveyor
event from_conveyor = start(Robot.after_turn==2 && robotFull_cond &&
        ((CONVEYOR_IOBUFFER && IOBufferEntryEmpty_cond) ||
         (CONVEYOR_CNC1 && cnc1EntryEmpty_cond) ||
          (CONVEYOR_CNC2 && cnc2EntryEmpty_cond) ||
          (CONVEYOR_CMM && cnc2EntryEmpty_cond)));
```

```
        event from_conveyor1 = from_conveyor when Robot.x_part_type==1;
        event from_conveyor2 = from_conveyor when Robot.x_part_type==2;
        event from_conveyor3 = from_conveyor when Robot.x_part_type==3;

        //a part has been put onto the conveyor
        event onto_conveyor = start(Robot.after_turn==3 && robotEmpty_cond &&
                ((IOBUFFER && IOBufferEntryFull_cond) ||
                (CNC1 && cnc1EntryFull_cond) ||
                (CNC2 && cnc2EntryFull_cond) ||
                (CMM && cnc2EntryFull_cond)));

        event onto_conveyor1 = onto_conveyor when Robot.x_part_type==1;
        event onto_conveyor2 = onto_conveyor when Robot.x_part_type==2;
        event onto_conveyor3 = onto_conveyor when Robot.x_part_type==3;
End
```

APPENDIX B

MEDL SCRIPT OF METUCIM LAB

```
ReqSpec METUCIM
        import condition part1_path, part2_path, part3_path;

        import event startPgm,
                    robotFull_event, robotEmpty_event,
                    robotFull_event1, robotEmpty_event1,
                    robotFull_event2, robotEmpty_event2,
                    robotFull_event3, robotEmpty_event3,
                    cnc1Full_event, cnc1Empty_event,
                    cnc1Full_event1, cnc1Empty_event1,
                    cnc1Full_event2, cnc1Empty_event2,
                    cnc1Full_event3, cnc1Empty_event3,
                    cnc2Full_event, cnc2Empty_event,
                    cnc2Full_event1, cnc2Empty_event1,
                    cnc2Full_event2, cnc2Empty_event2,
                    cnc2Full_event3, cnc2Empty_event3,
                    cmmFull_event, cmmEmpty_event,
                    cmmFull_event1, cmmEmpty_event1,
                    cmmFull_event2, cmmEmpty_event2,
                    cmmFull_event3, cmmEmpty_event3,
                    cnc1_start_work, cnc2_start_work, cmm_start_work,
                    cnc1_stop_work, cnc2_stop_work, cmm_stop_work,
                    IOBufferEntryFull_event, IOBufferEntryEmpty_event,
                    IOBufferEntryFull_event1, IOBufferEntryEmpty_event1,
                    IOBufferEntryFull_event2, IOBufferEntryEmpty_event2,
                    IOBufferEntryFull_event3, IOBufferEntryEmpty_event3,
                    cnc1EntryFull_event, cnc1EntryEmpty_event,
                    cnc1EntryFull_event1, cnc1EntryEmpty_event1,
                    cnc1EntryFull_event2, cnc1EntryEmpty_event2,
                    cnc1EntryFull_event3, cnc1EntryEmpty_event3,
                    cnc2EntryFull_event, cnc2EntryEmpty_event,
                    cnc2EntryFull_event1, cnc2EntryEmpty_event1,
                    cnc2EntryFull_event2, cnc2EntryEmpty_event2,
                    cnc2EntryFull_event3, cnc2EntryEmpty_event3,
                    from_conveyor, onto_conveyor, from_conveyor1, onto_conveyor1,
                    from_conveyor2, onto_conveyor2, from_conveyor3, onto_conveyor3,
          new_part, exit_part , exit_part1 , exit_part2 , exit_part3 ,
          control_start, control_end, abnormal_rotate,
          part1_path_event, part2_path_event, part3_path_event,
          CNC1_will_used, CNC2_will_used, CMM_will_used,
          CNC1_has_used, CNC2_has_used, CMM_has_used,
          wrong_IOBUFFER_process, wrong_CNC1_process,
          wrong_CNC2_process, wrong_CMM_process,
          wrong_CONVEYOR_IOBUFFER_process,
          wrong_CONVEYOR_CNC1_process,
          wrong_CONVEYOR_CNC2_process,
          wrong_CONVEYOR_CMM_process,
          new_part_type1, new_part_type2, new_part_type3,
          wrong_robot_place,endPgm;

//AuxilliaryVariables
var int SCALE;
var int part_count, part_in_cnt, part_out_cnt, exit_part_cnt;
var int part_count1, part_in_cnt1, part_out_cnt1, exit_part_cnt1;
var int part_count2, part_in_cnt2, part_out_cnt2, exit_part_cnt2;
var int part_count3, part_in_cnt3, part_out_cnt3, exit_part_cnt3;
var int part_in_type1_cnt;
var int part_in_type2_cnt;
var int part_in_type3_cnt;
var int robot_part_count;
var int robot_part_count1;
var int robot_part_count2;
var int robot_part_count3;
var int cnc1_part_count, cnc2_part_count, cmm_part_count;
var int cnc1_part_count1, cnc2_part_count1, cmm_part_count1;
var int cnc1_part_count2, cnc2_part_count2, cmm_part_count2;
var int cnc1_part_count3, cnc2_part_count3, cmm_part_count3;
var int iobuffer_entry_part_cnt,cnc1_entry_part_cnt,cnc2_entry_part_cnt;
var int iobuffer_entry_part_cnt1,cnc1_entry_part_cnt1,cnc2_entry_part_cnt1;
var int iobuffer_entry_part_cnt2,cnc1_entry_part_cnt2,cnc2_entry_part_cnt2;
var int iobuffer_entry_part_cnt3,cnc1_entry_part_cnt3,cnc2_entry_part_cnt3;
var int conveyor_part_count;
var int conveyor_part_count1;
var int conveyor_part_count2;
var int conveyor_part_count3;
var int cnc1_will_used, cnc2_will_used, cmm_will_used;
var int cnc1_has_used, cnc2_has_used, cmm_has_used;
var long time1;
var long time2;
var int exit_flag;

var long cnc1_time, cnc1_time1, cnc1_time2;
var long cnc2_time, cnc2_time1, cnc2_time2;
var long cmm_time, cmm_time1, cmm_time2;

//for calculating the time duration between two exit events
event exit_part_first      = exit_part  when exit_flag==1;
event exit_part_second = exit_part  when exit_flag==2;

//parts has been taken from the below components
```

```
event cnc1Empty = cnc1Empty_event when cnc1_time1>0;
event cnc2Empty = cnc2Empty_event when cnc2_time1>0;
event cmmEmpty = cmmEmpty_event when cmm_time1>0;

event cnc1Empty1 = cnc1Empty_event1 when cnc1_time1>0;
event cnc2Empty1 = cnc2Empty_event1 when cnc2_time1>0;
event cmmEmpty1 = cmmEmpty_event1 when cmm_time1>0;

event cnc1Empty2 = cnc1Empty_event2 when cnc1_time1>0;
event cnc2Empty2 = cnc2Empty_event2 when cnc2_time1>0;
event cmmEmpty2 = cmmEmpty_event2 when cmm_time1>0;

event cnc1Empty3 = cnc1Empty_event3 when cnc1_time1>0;
event cnc2Empty3 = cnc2Empty_event3 when cnc2_time1>0;
event cmmEmpty3 = cmmEmpty_event3 when cmm_time1>0;

//parts has been loaded into the below components
event cnc1Full = cnc1Full_event;
event cnc2Full = cnc2Full_event;
event cmmFull = cmmFull_event;

event cnc1Full1 = cnc1Full_event1;
event cnc2Full1 = cnc2Full_event1;
event cmmFull1 = cmmFull_event1;

event cnc1Full2 = cnc1Full_event2;
event cnc2Full2 = cnc2Full_event2;
event cmmFull2 = cmmFull_event2;

event cnc1Full3 = cnc1Full_event3;
event cnc2Full3 = cnc2Full_event3;
event cmmFull3 = cmmFull_event3;

//the below machines start and stop to process one part
event cnc1Stop = cnc1_stop_work when cnc1_time1>0;
event cnc2Stop = cnc2_stop_work when cnc2_time1>0;
event cmmStop = cmm_stop_work when cmm_time1>0;

event cnc1Start = cnc1_start_work;
event cnc2Start = cnc2_start_work;
event cmmStart = cmm_start_work;

//to control the time of transfering a part from one component to the other
//since at that time, it cannot known that to which component the part belongs
//two components can be seen full, or both can be seen empty at this time
//this violates the part count preservation rule
condition control = [control_start,control_end);


//Preservation of parts
property part_count_safe1 = !control || (part_in_cnt>=part_out_cnt);
property part_count_safe1_type1 = !control || (part_in_cnt1>=part_out_cnt1);
property part_count_safe1_type2 = !control || (part_in_cnt2>=part_out_cnt2);
property part_count_safe1_type3 = !control || (part_in_cnt3>=part_out_cnt3);
property part_count_safe2 = !control || (part_count==part_in_cnt-part_out_cnt);
property part_count_safe2_type1=!control ||
                                        (part_count1==part_in_cnt1-part_out_cnt1);
property part_count_safe2_type2=!control ||
                                        (part_count2==part_in_cnt2-part_out_cnt2);
property part_count_safe2_type3=!control ||
                                        (part_count3==part_in_cnt3-part_out_cnt3);

property part_count_safe3 = !control ||
            (part_count==robot_part_count+cnc1_part_count+cnc2_part_count+
                            cmm_part_count+conveyor_part_count);
property part_count_safe3_type1 = !control ||
            (part_count1==robot_part_count1+cnc1_part_count1+cnc2_part_count1+
                            cmm_part_count1+conveyor_part_count1);
property part_count_safe3_type2 = !control ||
            (part_count2==robot_part_count2+cnc1_part_count2+cnc2_part_count2+
                            cmm_part_count2+conveyor_part_count2);
property part_count_safe3_type3 = !control ||
            (part_count3==robot_part_count3+cnc1_part_count3+cnc2_part_count3+
                            cmm_part_count3+conveyor_part_count3);

alarm part_out_count_error = endPgm when (part_out_cnt!=part_in_cnt);
alarm part_out_count_error1 = endPgm when (part_out_cnt1!=part_in_cnt1);
alarm part_out_count_error2 = endPgm when (part_out_cnt2!=part_in_cnt2);
alarm part_out_count_error3 = endPgm when (part_out_cnt3!=part_in_cnt3);

alarm capasity_exceeded = start(part_count > 14);

//Request-process consistency
alarm wrong_IOBUFFER_process_alarm = wrong_IOBUFFER_process;
alarm wrong_CNC1_process_alarm = wrong_CNC1_process;
alarm wrong_CNC2_process_alarm = wrong_CNC2_process;
alarm wrong_CMM_process_alarm = wrong_CMM_process;
alarm wrong_CONVEYOR_IOBUFFER_process_alarm =
                                        wrong_CONVEYOR_IOBUFFER_process;
alarm wrong_CONVEYOR_CNC1_process_alarm =
                                        wrong_CONVEYOR_CNC1_process;
alarm wrong_CONVEYOR_CNC2_process_alarm =
                                        wrong_CONVEYOR_CNC2_process;
alarm wrong_CONVEYOR_CMM_process_alarm =
                                        wrong_CONVEYOR_CMM_process;

//part routes consistency
alarm wrong_part1_request = part1_path_event;
alarm wrong_part2_request = part2_path_event;
alarm wrong_part3_request = part3_path_event;

//robot sides
alarm robot_at_wrong_side = wrong_robot_place;
//time consistency
alarm cnc1_time_problem = endPgm when ((0.95*cnc1_time)>cnc1_time2 ||
                                                cnc1_time2>(1.05*cnc1_time));
alarm cnc2_time_problem = endPgm when ((0.95*cnc2_time)>cnc2_time2 ||
                                                      cnc2_time2>(1.05*cnc2_time));
alarm cmm_time_problem = endPgm when ((0.90*cmm_time)>cmm_time2 ||
                                                      cmm_time2>(1.10*cmm_time));

//event endPgm_for_cnc1 = endPgm when cnc1_time !=0;
event end_cnc1_time_cont = endPgm when ((0.95*cnc1_time)>cnc1_time2 ||
                                                      cnc1_time2>(1.05*cnc1_time));
event end_cnc2_time_cont = endPgm when ((0.95*cnc2_time)>cnc2_time2 ||
                                                      cnc2_time2>(1.05*cnc2_time));
event end_cmm_time_cont = endPgm when ((0.90*cmm_time)>cmm_time2 ||
                                                      cmm_time2>(1.10*cmm_time));

//minimum time between two consecutive exits
alarm min_time_difference = start(time1!=0 && time2!=0 &&
                                ((time1-time2 > 0 && time1-time2 < 30000/SCALE )||
                                (time2-time1 >0 && time2-time1 < 30000/SCALE )));

//visitation consistency
alarm CNC1_visitation_inconsistent=endPgm
                                           when (cnc1_will_used!=cnc1_has_used);
alarm CNC2_visitation_inconsistent=endPgm
                                           when (cnc2_will_used!=cnc2_has_used);
alarm CMM_visitation_inconsistent=endPgm
                                           when (cmm_will_used!=cmm_has_used);

//limited hardware capacity
alarm robot_capacity_exceeded  = start(robot_part_count>1);
alarm cnc1_capacity_exceeded   = start(cnc1_part_count>1);
alarm cnc2_capacity_exceeded   = start(cnc2_part_count>1);
alarm cmm_capacity_exceeded  = start(cmm_part_count>1);
alarm conveyor_capacity_exceeded  = start(conveyor_part_count>14);
alarm iobuffer_entry_capacity_exceeded = start(iobuffer_entry_part_cnt>1);
alarm cnc1_entry_capacity_exceeded      = start(cnc1_entry_part_cnt>1);
alarm cnc2_enrty_capacity_exceeded      = start(cnc2_entry_part_cnt>1);

//hardware is already empty
alarm robot_process_error = start(robot_part_count<0);
alarm cnc1_process_error = start(cnc1_part_count<0);
alarm cnc2_process_error = start(cnc2_part_count<0);
alarm cmm_process_error = start(cmm_part_count<0);
alarm conveyor_process_error = start(conveyor_part_count < 0);
alarm iobuffer_entry_already_empty = start(iobuffer_entry_part_cnt<0);
alarm cnc1_entry_already_empty = start(cnc1_entry_part_cnt<0);
alarm cnc2_entry_already_empty = start(cnc2_entry_part_cnt<0);

//conveyor rotation problem
alarm conveyor_rotate_problem = abnormal_rotate;

startPgm->{
          part_in_cnt=0;
          part_in_cnt1=0;
          part_in_cnt2=0;
          part_in_cnt3=0;
          part_in_type1_cnt=0;
```

```
part_in_type2_cnt=0;
part_in_type3_cnt=0;
part_out_cnt=0;
part_out_cnt1=0;
part_out_cnt2=0;
part_out_cnt3=0;
part_count=0;
part_count1=0;
part_count2=0;
part_count3=0;
exit_part_cnt=0;
exit_part_cnt1=0;
exit_part_cnt2=0;
exit_part_cnt3=0;
robot_part_count=1;
robot_part_count1=1;
robot_part_count2=1;
robot_part_count3=1;
cnc1_part_count=0;
cnc1_part_count1=0;
cnc1_part_count2=0;
cnc1_part_count3=0;
cnc2_part_count=0;
cnc2_part_count1=0;
cnc2_part_count2=0;
cnc2_part_count3=0;
cmm_part_count=0;
cmm_part_count1=0;
cmm_part_count2=0;
cmm_part_count3=0;
conveyor_part_count=0;
iobuffer_entry_part_cnt=0;
cnc1_entry_part_cnt=0;
cnc2_entry_part_cnt=0;
conveyor_part_count1=0;
iobuffer_entry_part_cnt1=0;
cnc1_entry_part_cnt1=0;
cnc2_entry_part_cnt1=0;
conveyor_part_count2=0;
iobuffer_entry_part_cnt2=0;
cnc1_entry_part_cnt2=0;
cnc2_entry_part_cnt2=0;
conveyor_part_count3=0;
iobuffer_entry_part_cnt3=0;
cnc1_entry_part_cnt3=0;
cnc2_entry_part_cnt3=0;
exit_flag=1;
time1=0;
time2=0;
cnc1_time=0;
cnc1_time1=0;
cnc1_time2=0;
cnc2_time=0;
cnc2_time1=0;
```

```
            cnc2_time2=0;
            cmm_time=0;
            cmm_time1=0;
            cmm_time2=0;
            cnc1_will_used=0;
            cnc2_will_used=0;
            cmm_will_used=0;
            cnc1_has_used=0;
            cnc2_has_used=0;
            cmm_has_used=0;
            SCALE=100;
}

new_part->{ part_in_cnt=part_in_cnt+1;
            part_count=part_count+1; }

new_part_type1->{ part_in_type1_cnt=part_in_type1_cnt + 1;
                  cnc1_time=cnc1_time+203000/SCALE;
                  cnc2_time=cnc2_time+109000/SCALE;
                  cmm_time=cmm_time+219000/SCALE;
                  cnc1_will_used=cnc1_will_used + 1;
                  cnc2_will_used=cnc2_will_used + 1;
                  cmm_will_used=cmm_will_used + 1;
                  part_in_cnt1=part_in_cnt1+1;
                  part_count1=part_count1+1; }

new_part_type2->{ part_in_type2_cnt=part_in_type2_cnt + 1;
                  cnc1_time=cnc1_time+103000/SCALE;
                  cmm_time=cmm_time+219000/SCALE;
                  cnc1_will_used=cnc1_will_used + 1;
                  cmm_will_used=cmm_will_used + 1;
                  part_in_cnt2=part_in_cnt2+1;
                  part_count2=part_count2+1; }

new_part_type3->{ part_in_type3_cnt=part_in_type3_cnt + 1;
                  cnc2_time=cnc2_time+131000/SCALE;
                  cmm_time=cmm_time+219000/SCALE;
                  cnc2_will_used=cnc2_will_used + 1;
                  cmm_will_used=cmm_will_used + 1;
                  part_in_cnt3=part_in_cnt3+1;
                  part_count3=part_count3+1; }

robotFull_event-> { robot_part_count=robot_part_count+1; }
robotEmpty_event->{ robot_part_count=robot_part_count-1;
                    robot_part_count1=0;
                    robot_part_count2=0;
                    robot_part_count3=0; }
robotFull_event1-> { robot_part_count1=robot_part_count1+1; }
robotFull_event2-> { robot_part_count2=robot_part_count2+1; }
robotFull_event3-> { robot_part_count3=robot_part_count3+1; }
cnc1Full-> { cnc1_part_count=cnc1_part_count+1; }
cnc1Empty->{ cnc1_part_count=cnc1_part_count-1;
             cnc1_has_used=cnc1_has_used+1;
             cnc1_part_count1=0;
```

```
                cnc1_part_count2=0;
                cnc1_part_count3=0;}
cnc1Full1-> { cnc1_part_count1=cnc1_part_count1+1; }
cnc1Full2-> { cnc1_part_count2=cnc1_part_count2+1; }
cnc1Full3-> { cnc1_part_count3=cnc1_part_count3+1; }

cnc1Start->{ cnc1_time1=time(cnc1Start); }
cnc1Stop  -> { cnc1_time2=cnc1_time2+(time(cnc1Stop)-cnc1_time1); }

cnc2Full  -> { cnc2_part_count=cnc2_part_count+1; }
cnc2Empty -> { cnc2_part_count=cnc2_part_count-1;
                 cnc2_has_used=cnc2_has_used+1;
                 cnc2_part_count1=0;
                 cnc2_part_count2=0;
                 cnc2_part_count3=0;}
cnc2Full1  -> { cnc2_part_count1=cnc2_part_count1+1; }
cnc2Full2  -> { cnc2_part_count2=cnc2_part_count2+1; }
cnc2Full3  -> { cnc2_part_count3=cnc2_part_count3+1; }

cnc2Start->{ cnc2_time1=time(cnc2Start); }
cnc2Stop->{ cnc2_time2=cnc2_time2+(time(cnc2Stop)-cnc2_time1); }

cmmFull->  { cmm_part_count=cmm_part_count+1; }
cmmEmpty->  { cmm_part_count=cmm_part_count-1;
                cmm_has_used=cmm_has_used+1;
                cmm_part_count1=0;
                cmm_part_count2=0;
                cmm_part_count3=0;}
cmmFull1  ->  { cmm_part_count1=cmm_part_count1+1; }
cmmFull2  ->  { cmm_part_count2=cmm_part_count2+1; }
cmmFull3  ->  { cmm_part_count3=cmm_part_count3+1; }

cmmStart->{ cmm_time1=time(cmmStart); }
cmmStop->{ cmm_time2=cmm_time2+(time(cmmStop)-cmm_time1); }

from_conveyor->{ conveyor_part_count=conveyor_part_count-1; }

from_conveyor1->{ conveyor_part_count1=conveyor_part_count1-1; }

from_conveyor2->{ conveyor_part_count2=conveyor_part_count2-1; }

from_conveyor3->{ conveyor_part_count3=conveyor_part_count3-1; }

onto_conveyor->{ conveyor_part_count=conveyor_part_count+1; }

onto_conveyor1->{ conveyor_part_count1=conveyor_part_count1+1; }
onto_conveyor2->{ conveyor_part_count2=conveyor_part_count2+1; }
onto_conveyor3->{ conveyor_part_count3=conveyor_part_count3+1; }

exit_part ->{ part_out_cnt=part_out_cnt+1;
                 part_count=part_count-1;
                exit_part_cnt=exit_part_cnt+1; }

exit_part1 ->{part_out_cnt1=part_out_cnt1+1;
```

```
                    part_count1=part_count1-1;
                    exit_part_cnt1=exit_part_cnt1+1; }

        exit_part2 ->{ part_out_cnt2=part_out_cnt2+1;
                       part_count2=part_count2-1;
                      exit_part_cnt2=exit_part_cnt2+1; }

        exit_part3 ->{ part_out_cnt3=part_out_cnt3+1;
                       part_count3=part_count3-1;
                      exit_part_cnt3=exit_part_cnt3+1; }

        exit_part_first ->{ time1=time(exit_part1);
                            exit_flag=2; }

        exit_part_second ->{ time2=time(exit_part2);
                              exit_flag=1; }

        IOBufferEntryFull_event  ->{ iobuffer_entry_part_cnt=iobuffer_entry_part_cnt+1; }
        IOBufferEntryEmpty_event->{ iobuffer_entry_part_cnt=iobuffer_entry_part_cnt-1;
                                    iobuffer_entry_part_cnt1=0;
                                    iobuffer_entry_part_cnt2=0;
                                    iobuffer_entry_part_cnt3=0; }
        cnc1EntryFull_event ->{   cnc1_entry_part_cnt=cnc1_entry_part_cnt+1; }
        cnc1EntryEmpty_event->{ cnc1_entry_part_cnt=cnc1_entry_part_cnt-1;
                               cnc1_entry_part_cnt1=0;
                               cnc1_entry_part_cnt2=0;
                               cnc1_entry_part_cnt3=0;}
        cnc2EntryFull_event ->{   cnc2_entry_part_cnt=cnc2_entry_part_cnt+1; }
        cnc2EntryEmpty_event->{ cnc2_entry_part_cnt=cnc2_entry_part_cnt-1;
                               cnc2_entry_part_cnt1=0;
                               cnc2_entry_part_cnt2=0;
                               cnc2_entry_part_cnt3=0;}

        IOBufferEntryFull_event1  ->{
                                iobuffer_entry_part_cnt1=iobuffer_entry_part_cnt1+1; }
        cnc1EntryFull_event1 ->{ cnc1_entry_part_cnt1=cnc1_entry_part_cnt1+1; }
        cnc2EntryFull_event1 ->{ cnc2_entry_part_cnt1=cnc2_entry_part_cnt1+1; }

        IOBufferEntryFull_event2  ->{
iobuffer_entry_part_cnt2=iobuffer_entry_part_cnt2+1; }
        cnc1EntryFull_event2  ->{ cnc1_entry_part_cnt2=cnc1_entry_part_cnt2+1; }
        cnc2EntryFull_event2  ->{ cnc2_entry_part_cnt2=cnc2_entry_part_cnt2+1; }

        IOBufferEntryFull_event3  ->{
iobuffer_entry_part_cnt3=iobuffer_entry_part_cnt3+1; }
        cnc1EntryFull_event3  ->{ cnc1_entry_part_cnt3=cnc1_entry_part_cnt3+1; }
        cnc2EntryFull_event3  ->{ cnc2_entry_part_cnt3=cnc2_entry_part_cnt3+1; }

End
```