UPE: Utah Prototyping Environment For Robot Manipulators

M. Dekhil, T. M. Sobh, T. C. Henderson and R. Mecklenburg
Department of Computer Science
University of Utah
Salt Lake City, Utah 84112.

Abstract. Developing an environment that enables optimal and flexible design of robot manipulators using reconfigurable links, joints, actuators, and sensors is an essential step for efficient robot design and prototyping. Such an environment should have the right "mix" of software and hardware components for designing the physical parts and the controllers, and for the algorithmic control of the robot modules (kinematics, inverse kinematics, dynamics, trajectory planning, analog control and digital computer control). Specifying object-based communications and catalog mechanisms between the software modules, controllers, physical parts, CAD designs, and actuator and sensor components is a necessary step in the prototyping activities.
In this paper, we propose a flexible prototyping environment for robot manipulators with the required subsystems and interfaces between the different components of this environment. This environment provides a close tie between the design parameters of the robot manipulator and the design constraints imposed by the required tasks and desired behaviors of the robot and by the different subsystems involved in the design process. The design and implementation of this environment along with the implementation of some of the subsystems are presented, and some examples that demonstrate the functionality of the environment are discussed. This work was supported in part by DARPA grant N00014-91-J-4123, NSF grant CDA 9024721, and a University of Utah Research Committee grant. All opinions, findings, conclusions or recommendations expressed in this document are those of the author and do not necessarily reflect the views of the sponsoring agencies.

Key words: Robot Design, Prototyping, Concurrent Engineering, Object-oriented Design

1. Introduction

Prototyping is an important activity in engineering. Prototype development is a good test for checking the viability of a proposed system. Prototypes can also help in determining system parameters, ranges, or in designing better systems. The interaction between several modules (e.g., S/W, VLSI, CAD, CAM, Robotics, and Control) illustrates an interdisciplinary prototyping environment that includes radically different types of information, combined in a coordinated way.

Our particular focus is in designing and building a robot manipulator. Many tasks are required, starting with specifying the performance requirements, determining the robot configuration and parameters that are most suitable for the required tasks, ordering the parts and assembling the robot, developing the necessary software and hardware components (controller, simulator, monitor), and finally, testing the robot and measuring its performance.

Our goal is to build a framework for optimal and flexible design of robot manipulators with software and hardware systems and modules which are independent of the design parameters and which can be used for different configurations and varying parameters. This environment is composed of several subsystems. Some of these subsystems are:

- Design.
- Simulation.
- Control.
- Monitoring.
- Hardware selection.
- CAD/CAM modeling.
- Part Ordering.
- Physical assembly and testing.

Each subsystem has its own structure, data representation, and reasoning strategy. On the other hand, much of the information is shared among these subsystems. To maintain the consistency of the whole system, an interface layer is proposed to facilitate the communication between these subsystems, and set the protocols that enable the interaction between the subsystems to take place. Figure 1 shows a schematic view of the prototyping environment with its subsystems and the interface.

This environment incorporates a unique constraint compilation and management scheme to ensure consistency of design changes across subsystems, and constitutes a major contribution of this paper.

A prototype 3-link robot manipulator was built to help determine the required sub-systems and interfaces to build the prototyping environment, and to provide hands-on experience for the real problems and difficulties that we would like to address and solve using this environment.

2. Background and Review

To integrate the work among different teams and sites working in such a large project, there must be some kind of synchronization to facilitate the communication and cooperation between them. A concurrent engineering infrastructure that encompasses multiple sites and subsystems, called Palo Alto Collaborative Testbed (PACT), was proposed in [5]. The issues discussed in that work were:

- Cooperative development of interfaces, protocols, and architecture.

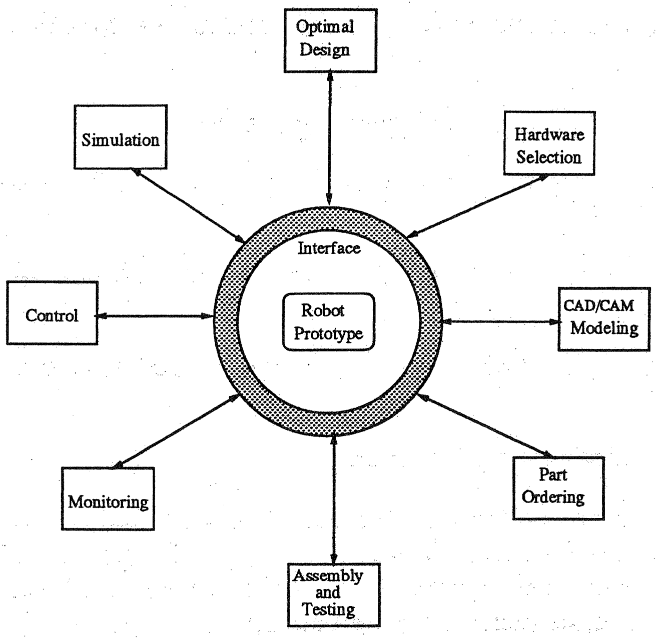


Fig. 1. Schematic view for the robot prototyping environment.

– Sharing of knowledge among heterogeneous systems.

– Computer-aided support for negotiation and decision-making.

An execution environment for heterogeneous systems called "InterBase" was proposed in [3]. It integrates preexisting systems over a distributed, autonomous, and heterogeneous environment via a tool-based interface. In this environment each system is associated with a *Remote System Interface (RSI)* that enables the transition from the local heterogeneity of each system to a uniform system-level interface. Our proposed system differs from this in that it incorporates a set of robotic design parameters and a set of constraints on these parameters. This provides a domain-specific knowledge that does not exist in the InterBase system.

Object orientation and its applications to integrate heterogeneous, autonomous, and distributed systems are discussed in [22]. The argument in this work is that object-oriented distributed computing is a natural step forward from the client-server systems of today. An automated, flexible and intelligent manufacturing based on object-oriented design and analysis techniques is discussed in [19], and a system for design, process planning and inspection is presented. We have used the same concept of object-oriented distributed design, but we have improved on this in UPE by adding the domain-specific constraints compiler as an essential component of the system which provides a convenient mean for providing and modifying the required design constraints.

Several important themes in concurrent software engineering are examined in [9]. Some of these themes are:

Tools: Specific tools that support concurrent software engineering.

Concepts: Tool-independent concepts are required to support concurrent software engineering.

Life cycle: Increase the concurrency of the various phases in the software life cycle.

Integration: Combining concepts and tools to form an integrated software engineering task.

Sharing: Defining multiple levels of sharing is necessary.

A management system for the generation and control of documentation flow throughout a whole manufacturing process is presented in [10]. The method of quality assurance is used to develop this system that covers cooperative work between different departments for documentation manipulation.

A computer-based architecture program called *the Distributed and Integrated Environment for Computer-Aided Engineering* (Dice), which addresses the coordination and communication problems in engineering, was developed at the MIT Intelligent Engineering Systems Laboratory [25]. The Dice project addresses several research issues such as, frameworks, representation, organization, design methods, visualization techniques, interfaces, and communication protocols. This is very similar to our project except for fact that UPE introduces a closer tie between the design parameters and the design constraints. The design constraints in our system are generated based on the required tasks and the desired behavior of the robot. These constraints are supplied either by a human expert, or by an expert system for optimal selection of robot parameters such as *TOCARD* [26]. Such systems can be integrated into the environment to support this task. Another difference is that in UPE there are two type of constraints; global constraints maintained by the central interface, and specific constraints maintained by each individual subsystem. This approach allows replacing any of the subsystems without modifying the global design constraints.

Some important topics in software engineering, such as the lifetime of a software system, analysis and design, module interfaces and implementation, and system testing and verification, can be found in [17]. Also, a report about integrated tools for product, and process design can be found in [27].

In the environment we are proposing for the design and prototyping of robot systems involving sensing and actuation, several subsystems communicate through a *central interface layer* (CI), and each subsystem has a *subsystem interface* (SSI) responsible for data transformation between the subsystem and the CI. The flexibility of this design arises from the following points:

- Adding new subsystem can be achieved by writing an SSI for the new subsystem, adding it to the list of the subsystems in the CI. There are no changes required to the other SSIs.
- Removing a subsystem only requires removing its name from the subsystems list in the CI.
- Any changes in one of the subsystems require changing the corresponding SSI to maintain correct data transformation to and from this subsystem.

3. Building a Three-link Robot

To explore the basis of building a flexible environment for robot manipulators, A three-link robot manipulator, "URK" (Utah Robot Kit), was designed. This enabled us determine the required subsystems and interfaces for such an environment. This prototype robot will be used as an educational tool in control and robotics classes.

This robot prototype can be easily connected to any workstation or PC through the standard serial port with an RS232 cable. Also, a controller for this robot was developed with an interface that enables the study of the manipulator's behavior for different design parameters and configurations. The manipulator was designed in such a way that enables the change of any of its sensors or actuators with minimal effort.

Figure 2 shows the physical three-link robot, and Figure 3 shows an overall view of the different interfaces and platforms that can control the robot. More details about this design can be found in [7, 24].

4. The Prototyping Environment

The proposed environment consists of several subsystems each of which carry out certain tasks to build the prototype robot. These subsystems share many parameters and information. To maintain the integrity and consistency of the whole system, a central interface (CI) is proposed with the required rules and protocols for passing information. This interface is the layer between the robot prototype and the subsystems, and it also serves as a communication channel between the different subsystems.

The difficulty of building such an interface arises from the fact that it deals with different systems, each with its own architecture, knowledge base, and reasoning mechanisms. In order to make these systems cooperate to maintain the consistency of the whole system, we have to understand the nature of the reasoning strategy for each subsystem, and the best way of transforming the information to and from each of them.

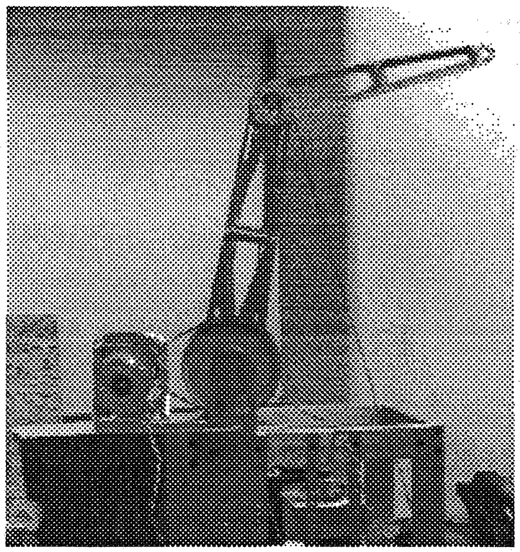


Fig. 2. The physical three-link robot manipulator.

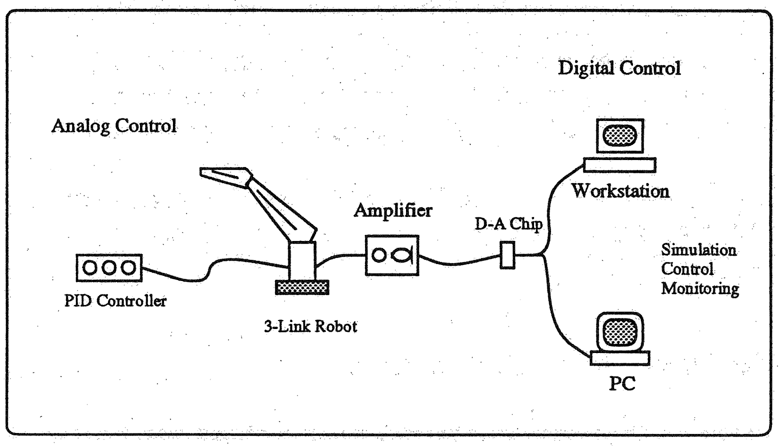


Fig. 3. Controlling the robot using different schemes.

In this environment the human role should be specified and a decision should be taken about which systems can be fully automated and which should be interactive with the user.

4.1. Overall Design

The Utah Prototyping Environment (UPE) consists of a *central interface* (CI) and *subsystem interfaces* (SSI). The tasks of the central interface are to:

- Maintain a global database of all the information needed for the design process.
- Communicate with the subsystems to update any changes in the system. This requires the central interface to know which subsystems need to know these changes and send messages to these subsystems informing them of the required changes.
- Receive messages and reports from the subsystems when any changes are required, or when any action has been taken (e.g., update complete).
- Transfer data between the subsystems upon request.
- Check constraints and apply some of the update rules.
- Maintain a design history containing the changes and actions that have been taken during each design process with date and time stamps.
- Deliver reports to the user with the current status and any changes in the system.

The subsystem interfaces are the interface layers between the CI and the subsystems. This makes the design more flexible and enables us to change any of the subsystems without much change in the CI — only the corresponding SSI need to be changed. The role of an SSI is:

- Report any changes to the CI.
- Receive messages from the CI with required updates.
- Perform the necessary updates in the actual files of the subsystem.
- Send acknowledgments or error messages to the CI.

The assumption is that there is a user at each subsystem (by a user here we mean one or more skilled persons who understand this subsystem), and there is a user operating the central interface as a general director and coordinator for the design process. In other words, the CI is to assist in the coordination of the job and to help communicate with all subsystems. Figure 4 shows an overall view of the suggested design.

In the first phase of implementing UPE, the users have more work to do. The CI and SSIs maintain the information routing between the subsystems by sending messages to the corresponding user at each subsystem, then the action itself (e.g., update a file) is accomplished by the user. Later on, the system will be automated to perform most of these actions itself and the user will simply be informed of the actions taken.

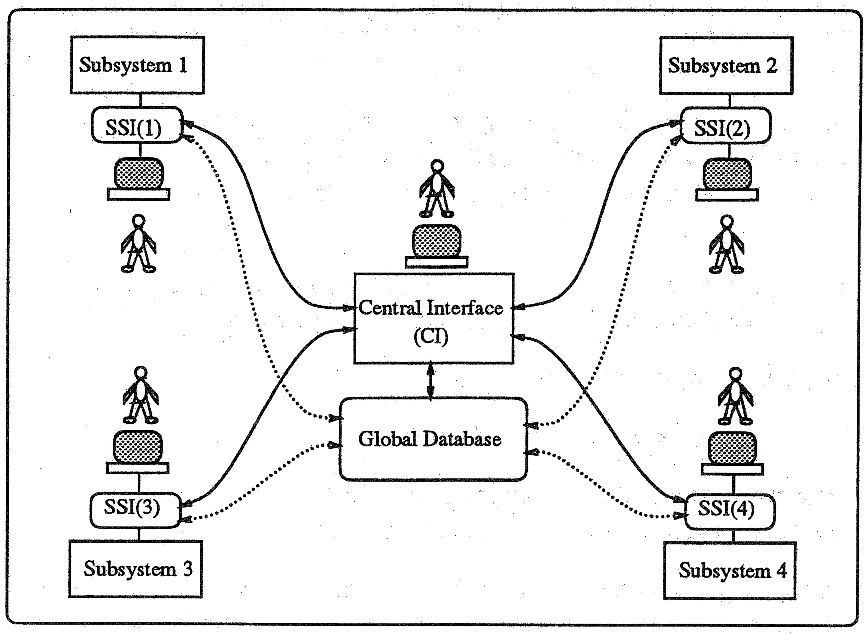


Fig. 4. Overall design of the prototyping environment.

4.2. Communication Protocols

The main purpose of this environment is keep all the subsystems informed of any changes in the design parameters. Therefore, passing information between the subsystems is the most important part of this environment. To be able to control the information flow, some protocols were developed to enable the communication between these subsystems in an organized manner. In our design, all subsystems communicate through the CI which is responsible for passing the information to the subsystems that need to know.

There are two types of events that can occur in this system:

1. Change reported from one of the subsystems.
2. Request for data from one subsystem to another.

Figure 5 shows the protocol used for the first event represented by a finite state machine (FSM). The states of this FSM are:

1. Steady state: Do nothing.
2. Change has been reported: send lock message to all subsystems. Apply relations and check constraints. If constraints are satisfied, go to state 3. If constraints are not satisfied, report these to sender and go to steady state.

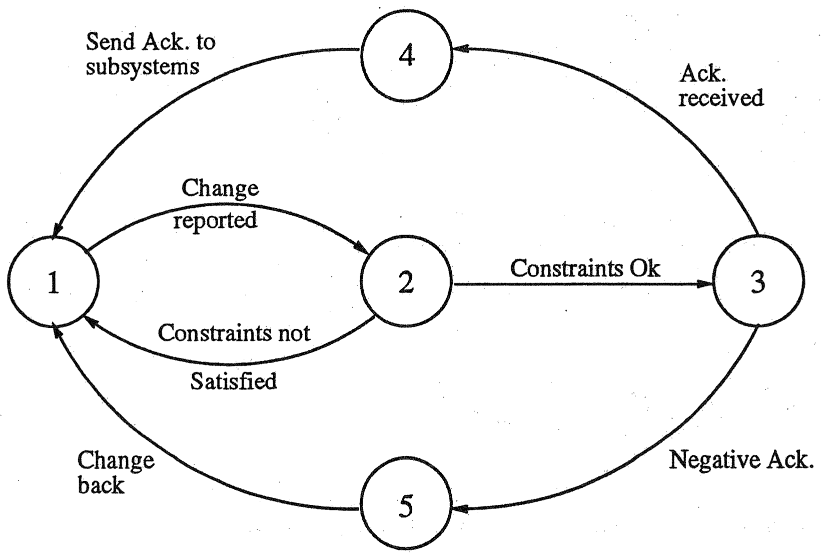


Fig. 5. Finite state machine representation for the change protocol.

3. Constraints are satisfied: Notify the subsystems with the changes and wait for acknowledgments.
4. Acknowledgments received from all subsystems: Send the final acknowledgment to the subsystems and go to steady state.
5. Acknowledgments not Ok: Send a "change-back" command to the subsystems and go to steady state.

Figure 6 shows the protocol for the second event. The states in this FSM are:

1. Steady state: Do nothing.
2. Request for S2 received from S1. Send the request to S2.
3. Required data found at S2. Send data to S1 and go to steady state.
4. Required data not found at S2. Send report to S1 and go to steady state.

The suggested protocol can be described in algorithmic notation as follows:

```
do while true
   if change reported then
       lock messages
       apply relations
       check constraints
```

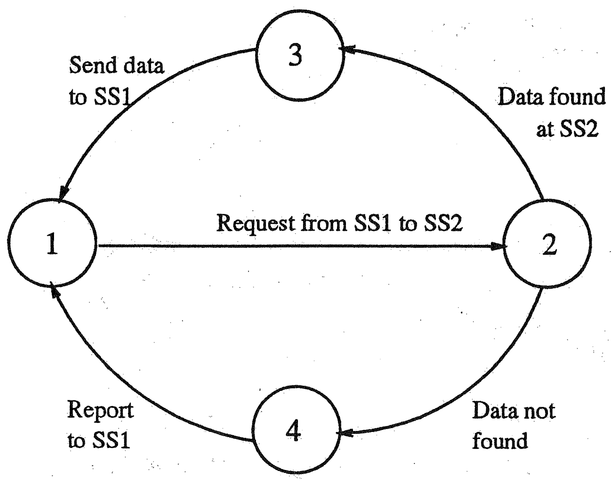


Fig. 6. Finite state machine representation for the data request protocol.

```
        if constraint satisfied then
            report changes to subsystems
            wait for subsystems acknowledgment
            if all acknowledgments ok
                update database
                report the new status
            else
                send a change-back message to subsystems
                report failure to sender
        else
            report nonsatisfied constraints to sender
        send final acknowledgment to subsystems
    else if data-request reported then
        send request to the appropriate subsystem
        if data received then
            send data to sender
        else
            send negative acknowledgment to sender.
```

Figure 7 shows a possible scenario when applying this protocol. In this algorithm we assume that all system constraints are located in the CI; however, any subsystem may reject the proposed values by other subsystems due to some unmodeled constraints. This can happen either because there are some "new" constraints that are not reported to the CI, or because some constraints are too hard to be easily represented in the constraint format in the CI.

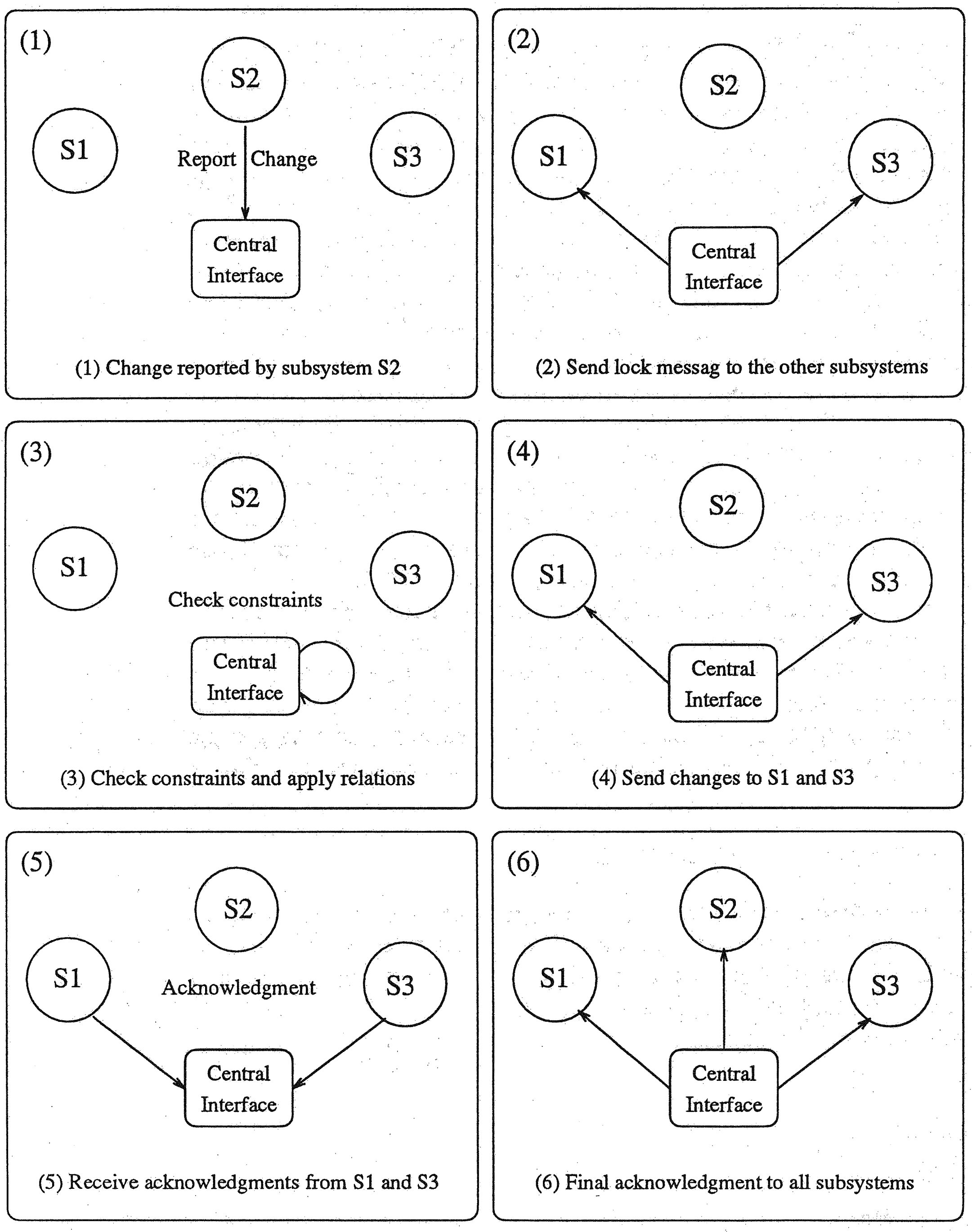

(1)
S2
S1
Report Change
S3
Central Interface
(1) Change reported by subsystem S2
(2)
S2
S1
S3
Central Interface
(2) Send lock messag to the other subsystems
(3)
S2
S1
S3
Check constraints
Central Interface
(3) Check constraints and apply relations
(4)
S2
S1
S3
Central Interface
(4) Send changes to S1 and S3
(5)
S2
S1
Acknowledgment
S3
Central Interface
(5) Receive acknowledgments from S1 and S3
(6)
S2
S1
S3
Central Interface
(6) Final acknowledgment to all subsystems

Fig. 7. Possible scenario for the communication between the subsystems.

4.3. Design Cycles and Infinite Loops

One problem that arises in UPE is that in some cases infinite design loops might occur due to some conflict between the constraints in different subsystems. For example, assume that the design system changed the link length to some value, say from 3.0 to 2.0 inches, to satisfy some performance requirements. This would alter the link mass as well, say from 1.5 to 1.0 lbs. According to the mass change the gear ratio has to change or the motor should be replaced, but if there is a constraint on the sprocket radius such that it can be increased, and there is no other motor with lower rpm, then the mass should be changed again to be 1.5 lbs, which requires the length to be 3.0 inches again. If we let the system continue, the design system will change the link length again and the loop will continue.

There are several solutions to this problem. One way is to make the user part of this loop so that some of the performance requirements can be changed, or a solution can be selected even if it does not meet some required criteria. This requires the user to be a skilled person who has the knowledge and experience in the design process, and also to have the authority to change and select solutions irrespective of the original requirements. Another solution is to put some limitations on the subsystem regarding its ability to change some of the design parameters. These limitations should guarantee infinite loop prevention in the system. A third solution is to put all the constraints in the CI. This allows the CI to check the solution and detect any violation to any of the constraints; then it may ask the user to decide on another solution or to change some of the performance requirements and run the design subsystem again. The last solution has the user in the loop as well, but incorporating all the constraints in the CI reduces the interprocess communication and speeds up the checking process. This last solution was chosen in our design.

4.4. Prototyping Environment Database

A database for the system components and the design parameters is necessary to enable the CI to check the constraints, to apply the update rules, to identify the subsystems that should be informed when any change happens in the system, and to maintain a design history and supply the required reports.

This database contains the following:

- Robot configuration.
- Design parameters.
- Available platforms.
- Design constraints.
- Subsystems information.
- Update rules.

– General information about the system.

Now the problem is to maintain this database. One solution is to use a database management system (DBMS) and integrate it in the prototyping environment. This requires writing an interface to transform the data from and to this DBMS, and this interface might be quite complicated. The other solution is to write our own DBMS. This sounds difficult, but we can make it very simple since the amount of data we have is limited and does not need sophisticated mechanisms to handle it. A relational database model is used in our design, and a user interface has been implemented to maintain this database. For the current design, by making a one-to-one correspondence between the classes and the files, reading and writing a file can be accomplished by adding member functions to each class. In this case there is no need for a special DBMS and all operations can be performed by simple functions.

4.5. Design Parameters

The design parameters are the most important data items in this environment. The main purpose of this system is to keep track of these parameters and notify the subsystems of any changes that occur to any of these parameters. For the system to perform this task, it needs to know the following data:

Figure 8 shows a list of the design parameters along with the subsystem that can change them and the subsystems that should be notified by a change in any of these parameters. Notice that some of these parameters are changed by the CI, and this change is accomplished using the update rules. In this figure note that one of the design parameters can be removed from this figure, which is "display rate." The removal of this parameter is valid because only one subsystem needs to know about this parameter and it is the same subsystem that can change it. However, we will keep it for possible future extensions or additions of other subsystems that might be interested in this parameter.

– A complete list of the design parameters.

– Which subsystems should be notified if a certain parameter is changed.

The optimal design subsystem is responsible for selecting most of the design parameters shown in Figure 8. The role of this subsystem is to assist robot designers in determining the optimal configuration and parameters given some task specifications and some of the parameters.

Designing an *optimal* manipulator is not yet well defined, and it depends on the definition and criterion of optimality. There are several techniques and methodologies to formalize this optimization problem by creating some objective functions that satisfy certain criteria, and solving these functions with the existence of some constraints. Some of the criteria that can be used to form objective functions are:

Design Parameter	CI	Design	Control	Simulation	Monitor	HW-Select	CAD/CAM	Ordering	Assembly
robot model	○	●	○	○	○		○		○
link length	○	●	○	○	○		○		○
link mass	●		○	○			○		○
link density	○	●					○		○
link cross area	○	●					○		○
joint friction	○	●	○	○			○		○
joint gear-ratio	●						○		○
update rate	○	●	○	○	○	○			
comm. rate	○	○	○	○		●			
motor rpm	○							●	○
motor range	○	●	○	○	○			○	○
sensor range	○	●	○	○	○	○		○	○
PID parameters	○	●	○	○					
display rate	○				●				
plateform	○				○	●			○

○ To be notified

● Make change

Fig. 8. Subsystem notification table according to parameter changes.

- Manipulability.
- Total motor power consumption.
- Arm weight.
- Total weight of robot.
- Cost.

- Workspace.
- Joint displacement limit.
- Maximum joint velocity and acceleration.
- Deflection.
- Natural frequency.
- Position accuracy.

To form the objective functions, we need to find quantitative measures for the manipulator specification and the performance requirements. In some cases, a closed form expression is not available. In such cases, the simulation programs can be used to determine the required quantitative measure. For example, the maximum velocity is a function of most of the parameters (link lengths, masses, friction, motor parameters), but it is not easy to get a closed form expression for the velocity as a function of all of these parameters; therefore, the simulation program can be used to measure the maximum velocity for different values of these parameters.

In addition to these quantitative measures, there are some rules and assumptions that can be used to solve for some of the parameters, and to give guidance during the design cycle. Some of the assumptions we made to simplify the problem are:

- The robot type and the degrees of freedom are given.
- Only revolute and prismatic joints are considered.
- The links are uniform with rectangular or cylindrical cross section.
- There is a finite set of materials used to build the robot with known densities.
- There is a finite number of actuators and sensors with known specifications that can be used in the design.

Some of the rules that can serve as additional constraints are:

- Select the solution with equal link lengths or masses because this will simplify the manufacturing process (i.e., minimize the cost).
- Choose the feedback controls k_p, k_v that give critically damped behavior ($k_v = 2\sqrt{k_p}$).
- Consider a minimum length for each link to satisfy some assembly and manufacturing constraints, such as actuator and sensors sizes.

Considerable research efforts has been done in this area. For example, The sue of kinematic criterion for the design evaluation of manipulators was investigated in [4, 20, 21, 16]. Another criterion is to achieve optimal dynamic performance; that is to select the link lengths and actuator sizes for minimum time motions along specified trajectory [18, 23].

TOCARD (Total Computer-Aided Design System of Robot Manipulators) is a system designed by Takano, Masaki, and Sasaki [26] to design both fundamental structure (degrees of freedom, arm length, etc.), and inner structure (arm size, motor allocation, motor power, etc).

Any of these techniques and systems can be incorporated in UPE by writing an SSI for that system and adding it as a new subsystem.

4.6. Database Design

A simple architecture for the database design is to make a one to one correspondence between classes and files; i.e., each file represents a class in the object analysis. For example, the robot file represents the robot class and each of the robot subclasses has a corresponding file. This design facilitates data transfer between the files and the system (the memory). On the other hand, this strong coupling between the database design and the system classes violates the database design rule of trying to make the design independent of the application; however, if the object analysis is done independently of the application intended, then this coupling is not a problem.

Now, we need to determine the format to be used to represent the database contents and the relations between the files in this database. Figure 9 shows the suggested data files that constitute the database for the system, and the data items in each file. The figure also shows the relations between the files. The single arrow arcs represent a one-to-one relation, and the double arrow arcs represent a one-to-many relation.

4.7. Constraints and Update Rules Compiler

A compiler is provided to generate C++ code for the constraints and the update rules. First, the syntax of the language that is used to describe the constraints and the update rules is described. Second, the generated code is determined.

Using a compiler instead of generic on-line evaluator for the constraints and the update rules has the following advantages:

- All constraints are saved in one text file (likewise the update rules). This makes the data entry very easy. We can add, update, and delete any constraint or update rule using any text editor.
- Complicated data structures are not required for evaluation.
- The database is very simple, which facilitates maintaining the design history.

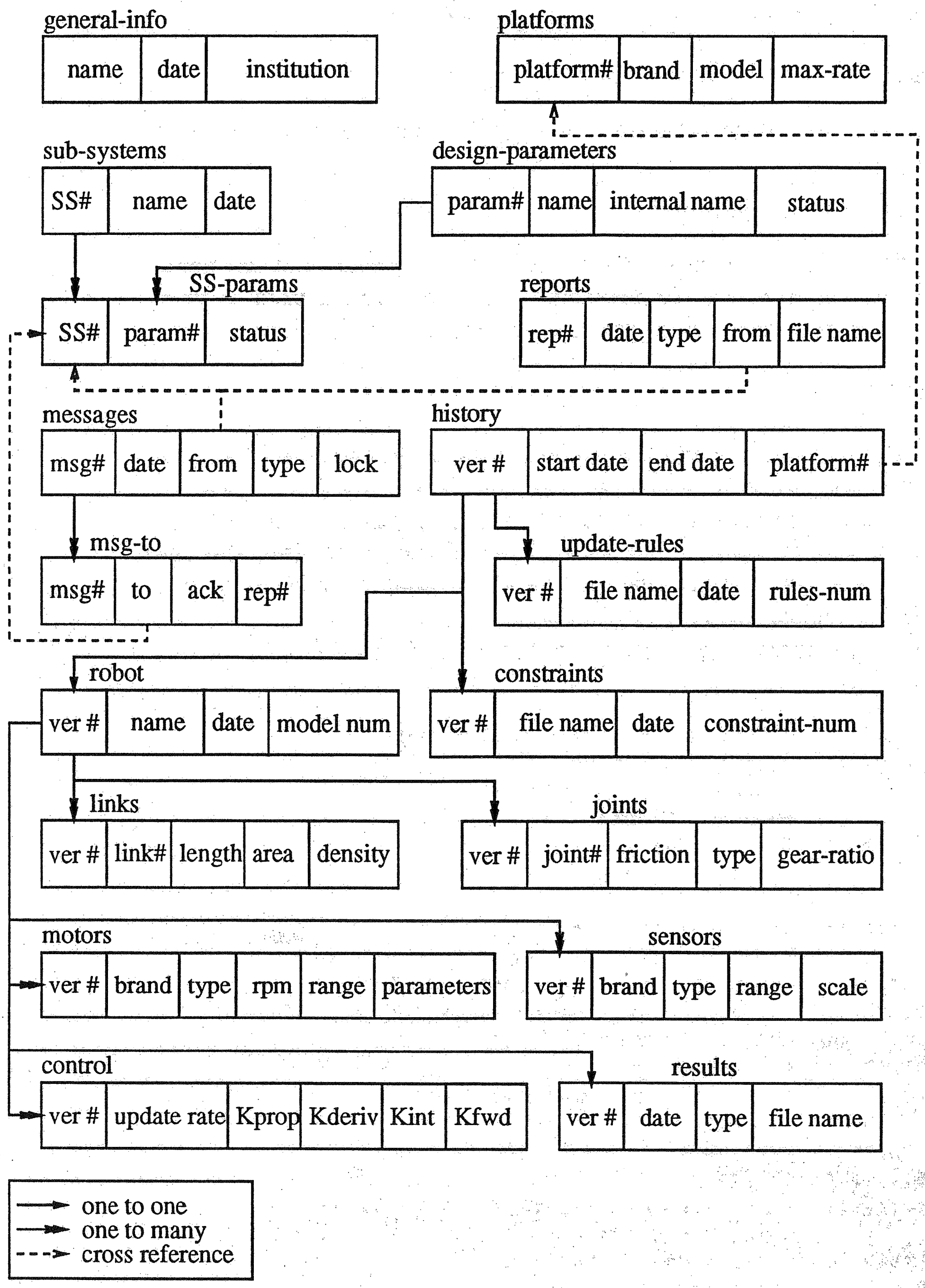


Fig. 9. Database design for the system.

– Format changes, or changes in the generated code require only changes to the compiler, and no changes in the system are required.

On the other hand, it has the following disadvantages:

– The generated code has to be included in the system and the whole system must be recompiled.

– A compiler needs to be implemented.

Notice that the changes in the constraints or the update rules are not frequent, so recompiling the system is not a big problem. Also, the syntax used is very simple; therefore the compiler for this language is not difficult to implement.

4.8. Language Syntax

By analyzing the design constraints and the update rules, we constructed a simple description of the language to be input to the compiler. There are two options in this design, either to have one compiler for both the constraints and the rules, or to build two compilers, one for each. From the analysis of the constraints and the rules we found that there are many similarities between them; thus building one compiler for both is the logical option in this case.

The following is the language definition in Backus Naur Form (BNF):

```
             <program> ::  <constraint-prog> | <rule-prog>
     <constraint-prog> ::  begin-constraints
                             <constraint-sequence>
                           end-constraints
           <rule-prog> ::  begin-rules
                             <rule-sequence>
                           end-rules
<constraint-sequence> ::  <constraint> ; <constraint-sequence> |
                           <constraint> ;
       <rule-sequence> ::  <rule> ; <rule-sequence> | <rule> ;
          <constraint> ::  <exp> <comparison-op> <exp>
                <rule> ::  <variable> = <exp>
                 <exp> ::  <exp> * <term> | <exp> / <term> | <term>
                <term> ::  <term> + <factor> | <term> - <factor> |
                           <factor>
              <factor> ::  <variable> | <constant> | (<exp>)
            <variable> ::  <alphabet> <alphanum> | <alphabet>
            <constant> ::  <int>.<int> | - <int>.<int> |
                           <int> | - <int>
                 <int> ::  <digit> <int> | <digit>
            <alphanum> ::  <alphabet> <alphanum> |
                           <digit> <alphanum> |
                           <alphabet> | <digit>
            <alphabet> ::  a..z | A..Z | _
          <digit> ::  0..9
  <comparison-op> ::  = | < | > | <= | >= | <>
```

The following is an example of some constraints described using this syntax:

```
begin-constraints
  link1_length > 1.2 ;
  link2_length > 1.5 ;
  link3_length > 0.8 ;
  link2_length + link3_length < MAX_TOT_LEN ;
  link1_mass < 1.4 ;
  link2_mass + link3_mass < 4.0 ;
  joint1_gear_ratio < 5.0 ;
end-constraints
```

Another example shows some update rules using the same syntax:

```
begin-rules
  link1_mass = link1_length * link1_density * link1_cross_area ;
  link2_mass = link2_length * link2_density * link2_cross_area ;
  link3_mass = link3_length * link3_density * link3_cross_area ;
  joint1_gear_ratio = motor1_speed / link1_max_speed ;
end-rules
```

From these examples it is clear that adding arrays to this language can reduce the length of the programs, but given the fact that these constraints and rules will be entered once at installation time, then adding or changing these rules and constraints will not be so frequent; thus, we will not complicate the compiler, at least in the first design phase. Some error detection and recovery modules for syntax error handling can be added to this compiler later.

4.9. The Generated Code

As mentioned before, this compiler generates C++ code which is integrated with the CI system to check the constraint or apply the update rule. Each variable in the input to the compiler corresponds to one design parameter. For example, "link1_length" corresponds to the variable in the CI system that represents the length of link number one in the robot configuration. The code generator uses a lookup table to find the corresponding variable name, and this table is part of the CI database. A simple flat file is used to store this table since the number of the design parameters is small.

The generated code for the constraints is the function "pe.check_constraints" that returns true if all constraints are satisfied, else it returns false, and reports which constraints are not satisfied. For the rules, the code generated is the function "pe.apply_rules" which calculates all corresponding design variables according to the given rules. The following examples are the code generated for the two examples shown in the previous section.

```
bool
ci::check_constraints()
{
    bool status[no_of_constraints] ;
    int i = 0 ;

    status[i++] = robot.configuration.link[0].length > 1.2 ;
    status[i++] = robot.configuration.link[1].length > 1.5 ;
    status[i++] = robot.configuration.link[2].length > 0.8 ;
    status[i++] = robot.configuration.link[1].length +
                  robot.configuration.link[2].length < 3.0 ;
    status[i++] = robot.configuration.link[0].mass < 1.4 ;
    status[i++] = robot.configuration.link[1].mass +
                  robot.configuration.link[2].mass  < 4.0 ;
    status[i]   = robot.configuration.joint[1].gear_ratio < 5.0 ;

    constraints.generate_report(status) ;   // report the result

    return (and_all(status)) ;
}


void
ci::apply_rules()
{
    robot.configuration.link[0].mass =
        robot.configuration.link[0].length *
        robot.configuration.link[0].cross_area *
        robot.configuration.link[0].density ;
    robot.configuration.link[1].mass =
        robot.configuration.link[1].length *
        robot.configuration.link[1].cross_area *
        robot.configuration.link[1].density ;
    robot.configuration.link[2].mass =
        robot.configuration.link[2].length *
        robot.configuration.link[2].cross_area *
        robot.configuration.link[2].density ;
    robot.configuration.joint[0].gear_ratio =
        robot.motor[0].speed /
        robot.configuration.joint[0].max_speed ;
}
```

In the first example, the function *generate_report* reports the results of checking the constraints; if all constraints are satisfied it reports that, otherwise, it will generate a list of the unsatisfied constraints. The function *and_all* returns the result of ANDing the elements in the array *status*.

In the second example, some of the design parameters are calculated given the values of some other parameters. The compiler should not allow the change of any parameter that should not be changed by the CI system. This can be detected using the *alter_flag* in the design parameters table.

To update the constraints or the update rules the file containing the old definition will be displayed and the user can add, delete, or update any of the old definitions. Then the new file will be compiled and integrated with the system.

5. Implementation

We have implemented this framework in order to prototype robotic systems. The subsystems include a CAD design system (Alpha_1 [11]), a robotic part-ordering system, a robot simulation package [6], and a robot control and monitoring system [8]. Figure 10 shows the graphical user interface used to control and monitor the three-link robot manipulator which was built as part of this project.

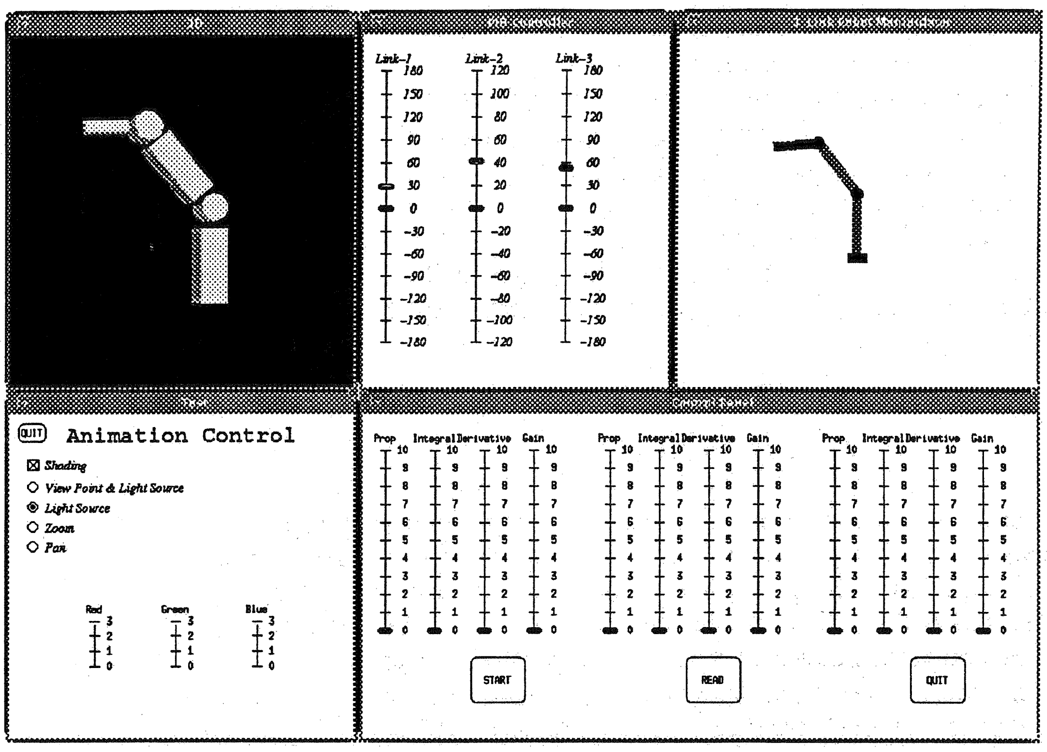
In the following subsections some implementation issues are investigated, and the different components in our design and how we implemented each of them are described.

5.1. The Central Interface

The central-interface (CI) is the core program that handles the communication between the subsystems, and maintains a global database for the current design and a history of previous designs. There are several types of messages used in the communication. Table I shows the different types of messages with a brief description and the direction of each.

The CI is the implementation of the communication protocols described in Section 4.2, with some enhancements. For example, when the CI receives a *change* message from an SSI, it directly sends lock messages to the other subsystems so that no more changes can be sent from any SSI until they receive a *steady* message. This solves the concurrency problem of more than one system sending changes to the CI at the same time. The first message received by the CI will be handled and the others will be ignored. If an SSI receives a *lock* message after it sent a *change* message, that means its message was ignored. Another feature added to the CI is the ability to detect if an SSI is working or not by tracing the *SSI_Start* and *SSI_Stop* messages.

The CI is managing a number of data files that contain information about the robot configuration, platforms, reports, design history, subsystems, and some general information about the project. The basic file operation was implemented



Animation Control

☒ Shading
○ View Point & Light Source
● Light Source
○ Zoom
○ Pan

Fig. 10. The Interface Window for the PID Controller Simulator.

by defining a file class, and by adding some member functions to each class in the system that performs the required file management operations.

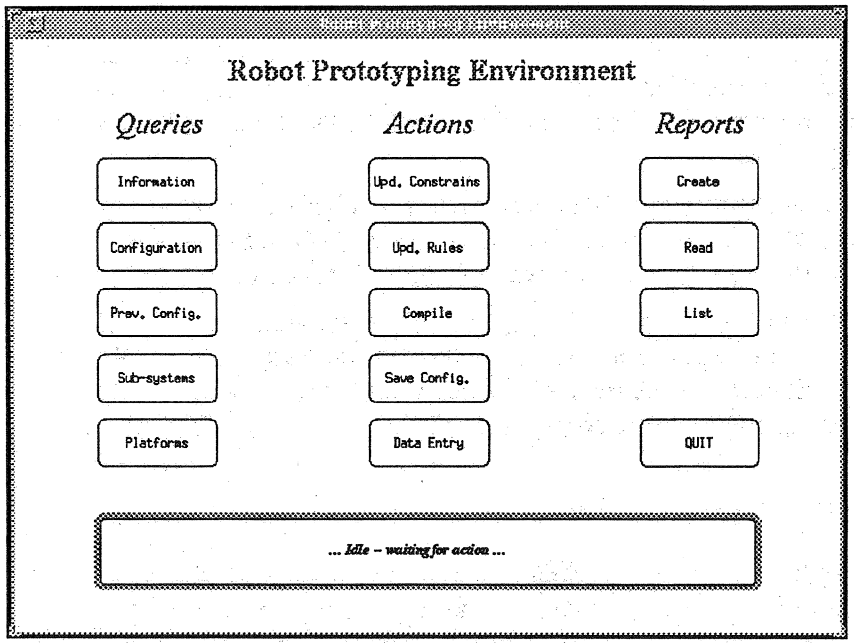
5.2. The PE Control System

The CI as described above has no user interface. To be able to control and manage the coordination between the subsystems, the PE control system (PECS) was implemented with some functionalities that enable the user to have some control over the CI.

The PECS is on top of the simple DBMS and a simple compiler for the update rules and the constraints. The user specifies the constraints and/or the update rules using a certain format (a language), then the compiler transforms this to C code that will be integrated with the system for constraint checking, and for applying the update rules. The compiler consists of two parts, a parser and a code generator. In the first phase the complexity of the compiler was reduced by making the user language less sophisticated. Later on this can be easily replaced by a more complicated compiler with an easier interface and more sophisticated error checking and optimization capabilities. Figure 11 shows the user interface for the PECS.

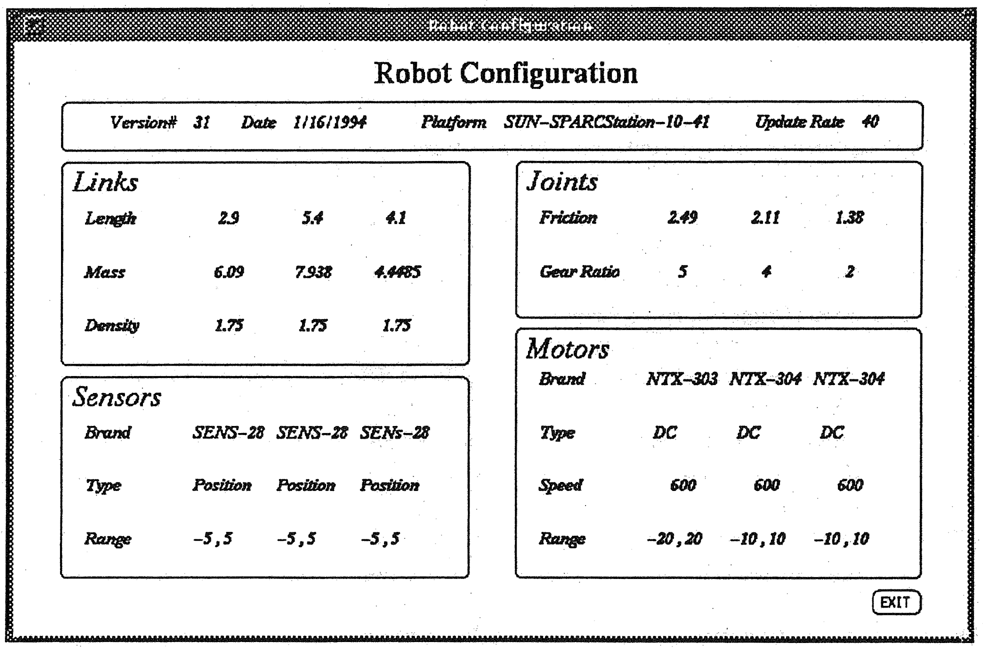
TABLE I
Message types used in the communication protocols.

Type	Description	Direction
Change	Data change reported	SSI $\longrightarrow$ CI
Const_Not_Ok	Constraints not satisfied	CI $\longrightarrow$ SSI
Notify	Send changes to subsystems	CI $\longrightarrow$ SSI
Ack.	Positive acknowledgment	SSI $\longrightarrow$ CI
Neg_Ack.	Negative acknowledgment	SSI $\longrightarrow$ CI
Back	Change back	CI $\longrightarrow$ SSI
Steady	Final acknowledgment	CI $\longrightarrow$ SSI
Request	Request for data	CI $\longleftrightarrow$ SSI
Found	Data found	CI $\longleftrightarrow$ SSI
Not_Found	Data not found	CI $\longleftrightarrow$ SSI
Lock	lock messages	CI $\longrightarrow$ SSI
SSI_Start	SSI is activated	SSI $\longrightarrow$ CI
SSI_Stop	SSI is terminated	SSI $\longrightarrow$ CI
Terminate	Terminate the CI.	UPE control $\longrightarrow$ CI



... Idle – waiting for action ...

Fig. 11. The main window for the PE control system.


Robot Configuration

Version# 31 Date 1/16/1994 Platform SUN-SPARCStation-10-41 Update Rate 40

Links

Length	2.9	5.4	4.1
Mass	6.09	7.938	4.4485
Density	1.75	1.75	1.75

Joints

Friction	2.49	2.11	1.38
Gear Ratio	5	4	2

Sensors

Brand	SENS-28	SENS-28	SENs-28
Type	Position	Position	Position
Range	-5,5	-5,5	-5,5

Motors

Brand	NTX-303	NTX-304	NTX-304
Type	DC	DC	DC
Speed	600	600	600
Range	-20,20	-10,10	-10,10

Fig. 12. The current robot configuration window.

The PECS functions include:

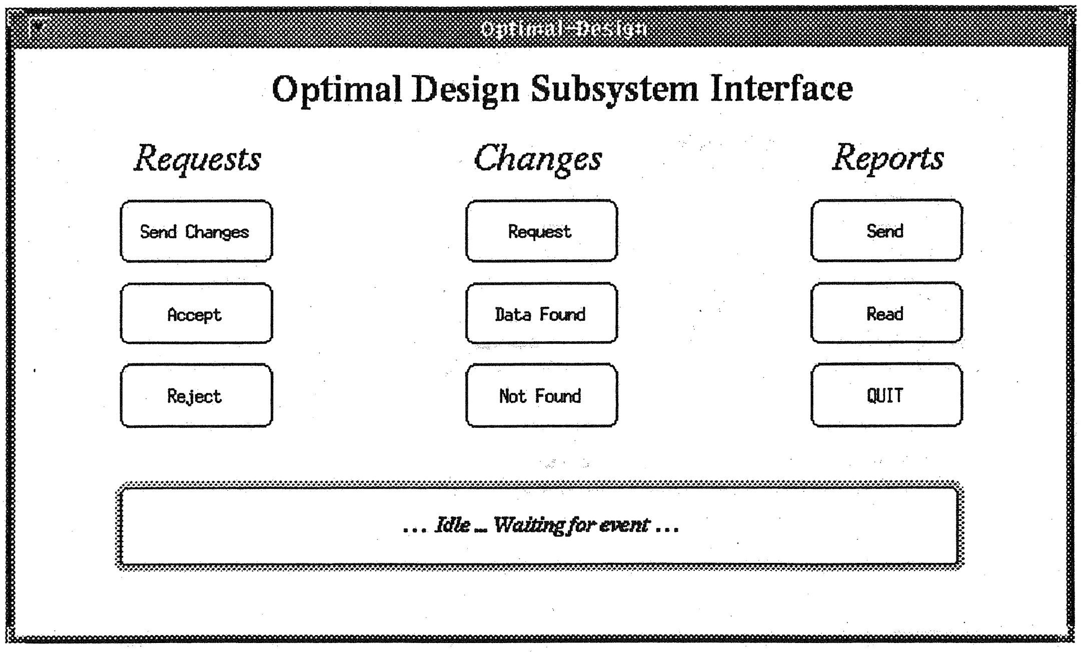
Queries: these are simple reports about the current robot configuration, previous configuration, general information about the system, the platforms, and the subsystems of the prototyping environment. Figure 12 shows a query for the current robot configuration.

Actions: these are the actual operations that control the CI. These actions include updating the constraints and the update rules, compiling the CI after including the new constraints and update rules, activating, and terminating the CI.

Reports: these are operations to manage the reports in the system, and to send and receive reports to and from the subsystems. The report can be text, graph, figure, postscript, or data file. Each report is saved with its type, date, sender, and the file that contains the report contents.

5.3. Initial Implementation of the SSIs

In the first phase of implementation, the SSIs serve as a simple interface layer between the CI and the user at each subsystem. They receive messages from the CI and display them to the user who takes any necessary actions. They also report any changes to the CI, and this is done by sending a message to the CI with the changes. Figure 13 shows the user interface for one of the SSIs.



... Idle ... Waiting for event ...

Fig. 13. The user interface for the SSI.

In the next implementation phase, some of the actions will be automated and the user at each subsystem will be notified with any action taken. For example, updating a data file that is used by the subsystem can be automatically done by the SSI, given that it has the necessary information about the file format and the location of the changed data.

5.4. The Central Interface Monitor

The central interface monitor (CIM) enables the user to monitor the actions and the messages passing between the CI and the SSIs with a graphical interface. This interface shows the CI in the middle and the SSIs as small boxes surrounding the CI. The CIM also has a small text window at near the bottom. This text window describes the current action (see Figure 14). The messages are represented by an arrow from the sender to the receiver. Some results of testing the CI and the SSIs are presented in Section 6 with sequences of the CIM window showing the activities that took place in each experiment.

6. Results

In this section, we will show several test cases for the prototyping environment. In the first test (Figure 15), the optimal design subsystem sent a data-change message

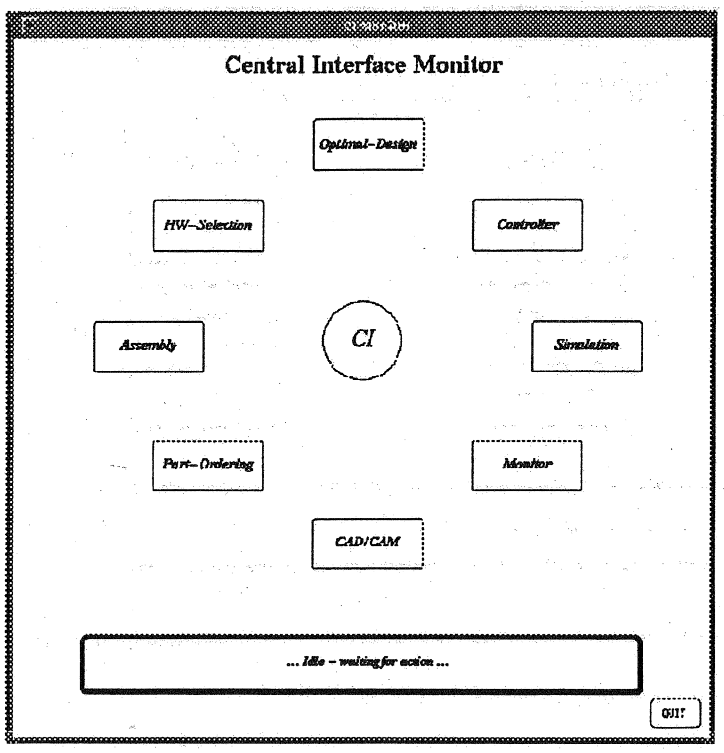


Fig. 14. The graphical interface for the monitor system.

to the CI. The CI in turn sent lock messages to all other subsystems notifying them that no changes will be accepted until they receive a final acknowledgment message. Then, the CI applied the relations and checked the design constraints. In this test case the constraints were satisfied, so the CI sent these changes to the subsystems that needed to be notified. After that, the CI waited for acknowledgments from the subsystems. In this case it received positive acknowledgments from the specified subsystems. Finally, the CI updated the database and sent final acknowledgment messages to all subsystems.

The second test case (Figure 16), was the same as the first case except that one of the subsystems (the CAD/CAM subsystem) rejected the changes by sending a negative acknowledgment message to the CI. Thus, the CI sent a change-back message to the specified subsystems and then sent a final acknowledgment message to all subsystems. No changes in the database took place in this case.

In the last test case (Figure 17), the design constraints were not satisfied. Therefore, the CI sent a report about the nonsatisfied constraints to the sender (the optimal design subsystem). Then it sent final acknowledgment messages to all subsystems. Again, in this case, no changes in the database took place.

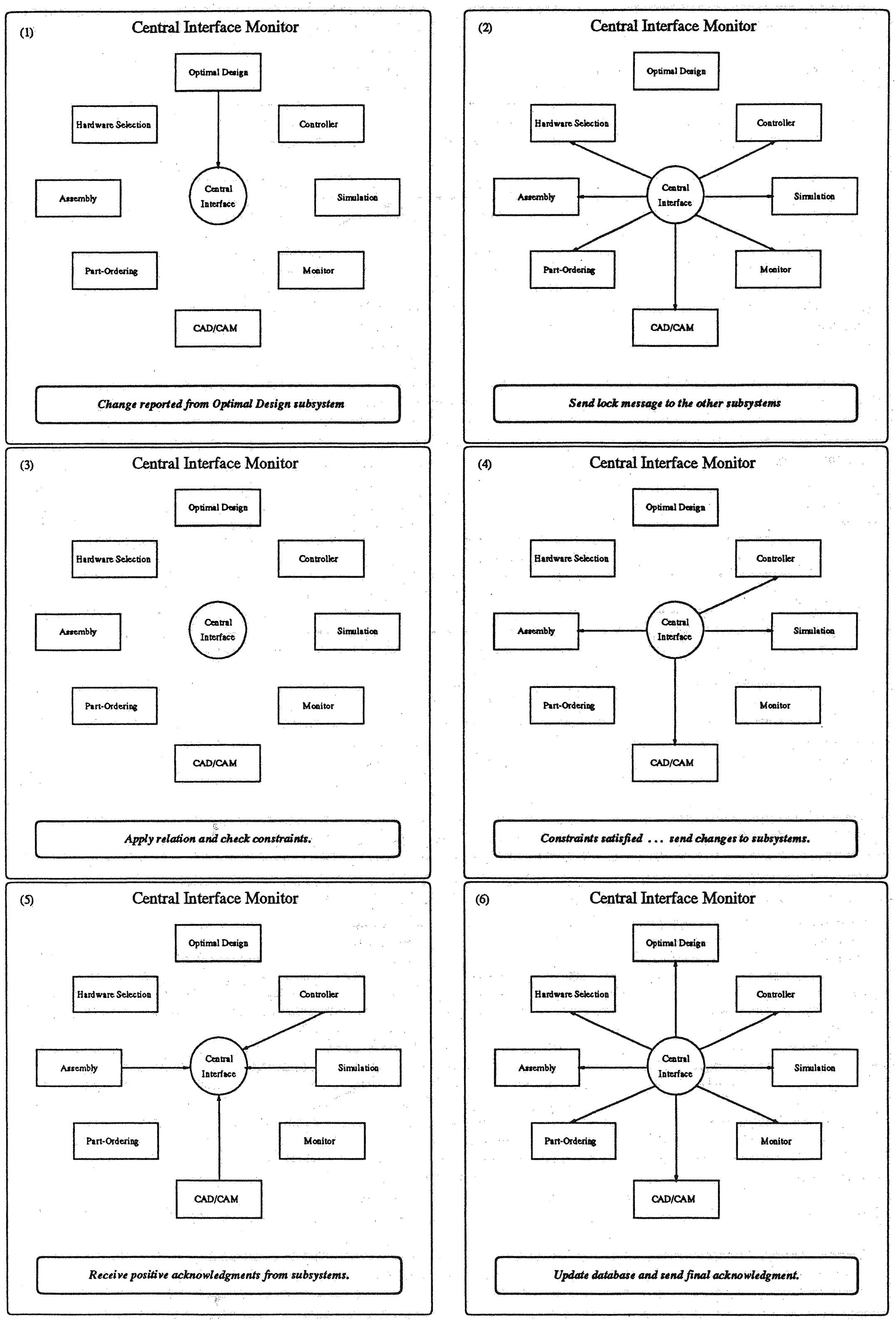

(1)
Central Interface Monitor
Optimal Design
Hardware Selection
Controller
Assembly
Central Interface
Simulation
Part-Ordering
Monitor
CAD/CAM
Change reported from Optimal Design subsystem
(2)
Central Interface Monitor
Send lock message to the other subsystems
(3)
Central Interface Monitor
Apply relation and check constraints.
(4)
Central Interface Monitor
Constraints satisfied ... send changes to subsystems.
(5)
Central Interface Monitor
Receive positive acknowledgments from subsystems.
(6)
Central Interface Monitor
Update database and send final acknowledgment.

Fig. 15. CI test case one, success case for data change.

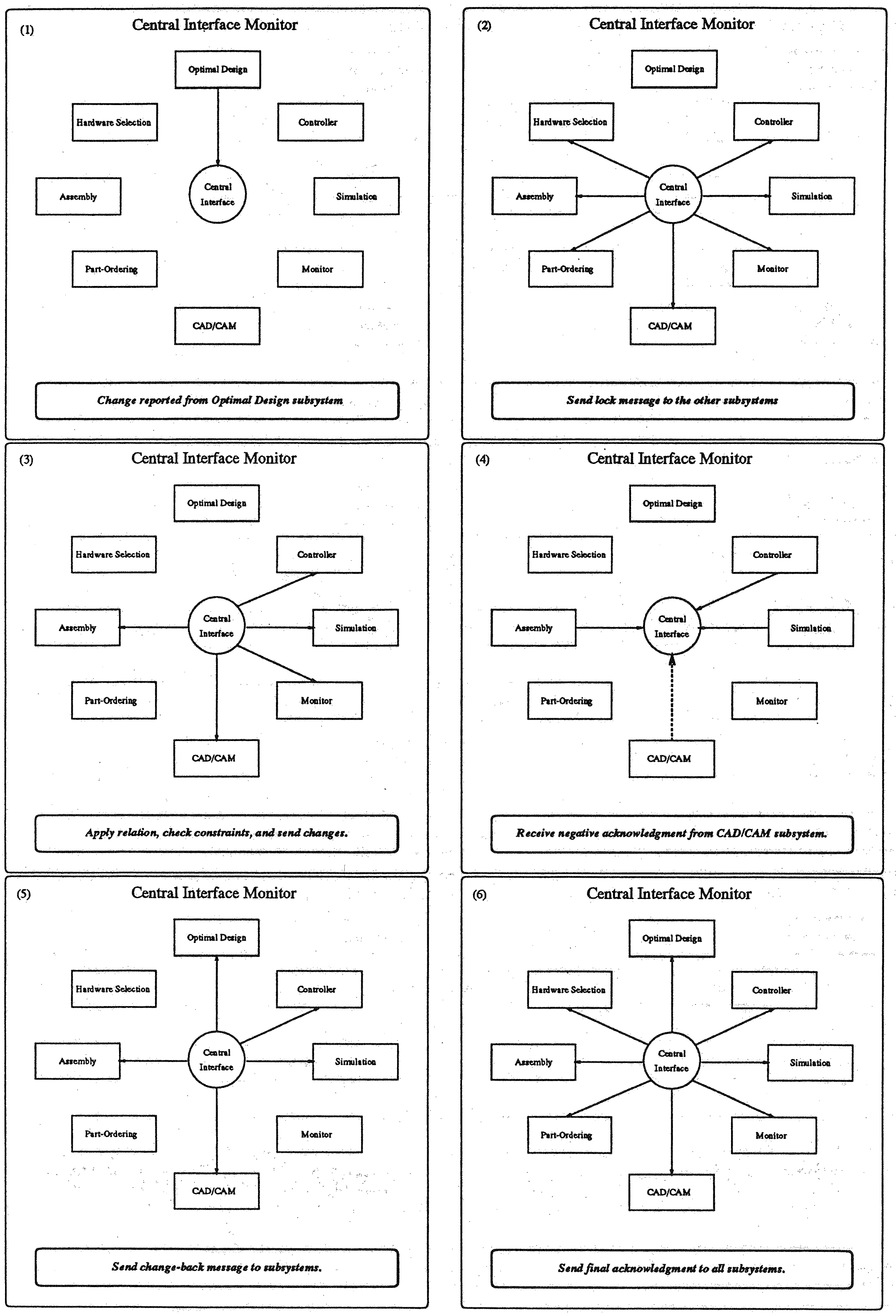


Fig. 16. CI test case two, negative acknowledgment case.

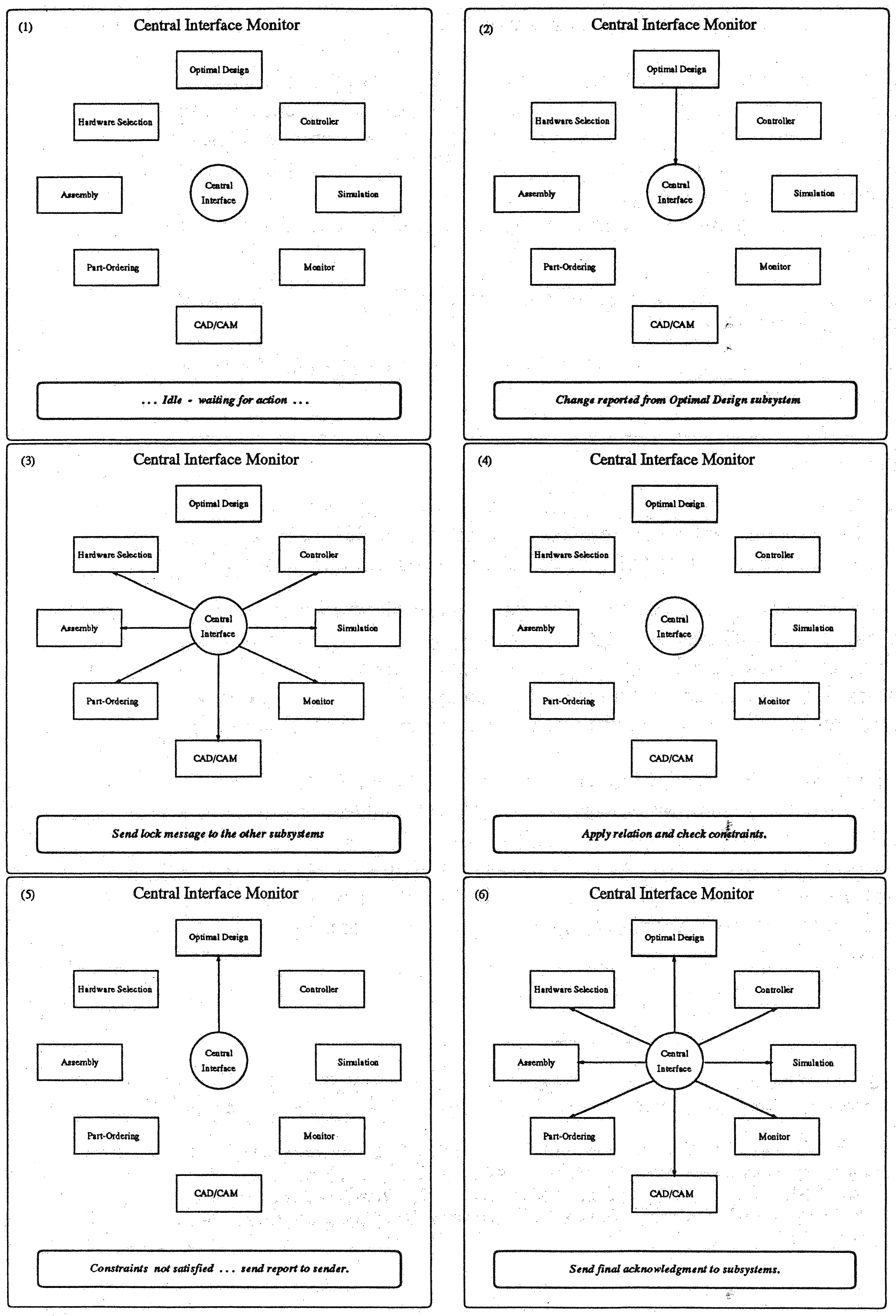

(1)
Central Interface Monitor
Optimal Design
Hardware Selection
Controller
Assembly
Central Interface
Simulation
Part-Ordering
Monitor
CAD/CAM
... Idle - waiting for action ...
(2)
Central Interface Monitor
Change reported from Optimal Design subsystem
(3)
Central Interface Monitor
Send lock message to the other subsystems
(4)
Central Interface Monitor
Apply relation and check constraints.
(5)
Central Interface Monitor
Constraints not satisfied ... send report to sender.
(6)
Central Interface Monitor
Send final acknowledgment to subsystems.

Fig. 17. CI test case three, nonsatisfied constraints case.

7. Conclusions and Future Work

The design basis for building a prototyping environment for robot manipulators was investigated and the design options were explained. An initial implementation of a central interface and some of the subsystem interfaces was done to demonstrate the functionality of the proposed environment. This framework facilitates and speeds the design process of robots.

The following are some possible extensions and enhancements to the current design.

- Complete implementation for the central interface with more functionality and a user friendly interface.
- Use a database query language to enable generating more sophisticated queries and to enhance the report generating capabilities.
- Implement some of the subsystems with their SSIs and increase the automation in these interfaces.
- Extend this environment to deal with generic *n*-link robots by using automatic generation of the kinematics and dynamics equations. Also this will require a robot description language to specify the robot configuration and parameters.

We have done a lot of work on robot behavior specification and analysis [1, 6, 12, 14, 13, 2]. One opportunity to tremendously increase the scope of the prototyping system we would like to explore is the use of a behavior analysis subsystem as part of the design feedback in order to constrain the system. Thus, a complete system can be tested in a simulated environment and the results used to modify the design.

Another area of interest is the use of information from the manufacturing side of the prototyping endeavor. Oftentimes changes are made to expedite the production of the system (e.g., mechanical parts are modified to make machining easier). We have done a good bit of work on reverse engineering of mechanical parts[15], and believe that is will be useful to feedback to UPE details concerning such changes. Their impact can be determined by propagating the results to all the other subsystems.

References

1. Bradakis, M., Henderson, T. C., and Zachary, J. Reactive behavior design tools. In *IEEE International Symposium Intelligent Control* (Glasgow, Scotland, 1992), Phantom House.
2. Bradakis, M., Henderson, T. C., and Zachary, J. Reactive behavior design tools. In *International Symposium on Intelligent Control* (Glasgow, Scotland, 1992), pp. 173–183.
3. Bukhres, O. A., Chen, J., Du, W., and Elmagarmid, A. K. Interbase: An execution environment for heterogeneous software systems. *IEEE Computer Magazine* (Aug. 1993), pp. 57–69.

4. CHIU, S. L. Kinematic characterization of manipulators: An approach to defining optimality. In *IEEE Int. Conf. Robotics and Automation* (1988), pp. 828–833.
5. CUTKOSKY, M. R., ENGELMORE, R. S., FIKES, R. E., GENESERETH, M. R., GRUBER, T. R., MARK, W. S., TENENBAUM, J. M., AND WEBER, J. C. PACT: An experiment in integrating concurrent engineering systems. *IEEE Computer Magazine* (Jan. 1993), pp. 28–37.
6. DALTON, P. J. Z-infinity: A framework for reactive autonomous agent specification and analysis. Master's thesis, University of Utah, Salt Lake City, Utah, 1994.
7. DEKHIL, M., SOBH, T. M., AND HENDERSON, T. C. URK: Utah Robot Kit - a 3-link robot manipulator prototype. In *IEEE Int. Conf. Robotics and Automation* (May 1994).
8. DEKHIL, M., SOBH, T. M., HENDERSON, T. C., AND MECKLENBURG, R. Robotic prototyping environment (progress report). Tech. Rep. UUCS-94-004, University of Utah, Feb. 1994.
9. DEWAN, P., AND RIEDL, J. Toward computer-supported concurrent software engineering. *IEEE Computer Magazine* (Jan. 1993), pp. 17–27.
10. DUHOVNIK, J., TAVCAR, J., AND KOPOREC, J. Project manager with quality assurance. *Computer-Aided Design 25*, 5 (May 1993), pp. 311–319.
11. ENGINEERING GEOMETRY SYSTEMS. *Alpha_1 Programmer's Manual*, 1992.
12. GRUPEN, R., AND HENDERSON, T. C. Autochthonous behaviors: Mapping perception to action. In *NATO ASI on Traditional and Non-Traditional Robotic Sensors* (Heidelberg, West Germany, 1990), T. C. Henderson, Ed., Springer-Verlag, pp. 285–312.
13. HENDERSON, T. C., DALTON, P., AND ZACHARY, J. A research program for autonomous agent behavior specification and analysis. In *IEEE International Symposium on Intelligent Control* (Washington, D.C, 1991).
14. HENDERSON, T. C., AND GRUPEN, R. Logical behaviors. *Journal of Robotic Systems 7*, 3 (1990), 309–336.
15. HENDERSON, T. C., AND THOMPSON, W. B. Image understanding research at the university of utah. In *ARPA 1994 IU Workshop* (Monterey, CA, 1994).
16. HOLLERBACH, J. Optimum kinematic design for a seven degree of freedom manipulator. In *Robotics Research: 2nd Int. Symp.* (1985), H. Hanafusa and H. Inous, Eds., MIT Press, pp. 215–222.
17. LAMB, D. A. *Software Engineering: Planning for Change.* Prentice Hall, 1988.
18. MA, O., AND ANGELES, J. Optimum design of manipulators under dynamic isotropy conditions. In *IEEE Int. Conf. Robotics and Automation* (1993), pp. 470–475.
19. MAREFAT, M., MALHORTA, S., AND KASHYAP, R. L. Object-oriented intelligent computer-integrated design, process planning, and inspection. *IEEE Computer Magazine* (Mar. 1993), pp. 54–65.
20. MAYORGA, R. V., RESSA, B., AND WONG, A. K. C. A kinematic criterion for the design optimization of robot manipulators. In *IEEE Int. Conf. Robotics and Automation* (1991), pp. 578–583.
21. MAYORGA, R. V., RESSA, B., AND WONG, A. K. C. A kinematic design optimization of robot manipulators. In *IEEE Int. Conf. Robotics and Automation* (1992), pp. 396–401.
22. NICOL, J. R., WILKES, C. T., AND MANOLA, F. A. Object orientation in heterogeneous distributed computing systems. *IEEE Computer Magazine* (June 1993), pp. 57–67.
23. SHILLER, Z., AND SUNDAR, S. Design of robot manipulators for optimal dynamic performance. In *IEEE Int. Conf. Robotics and Automation* (1991), pp. 344–349.
24. SOBH, T. M., DEKHIL, M., AND HENDERSON, T. C. Prototyping a robot manipulator and controller. Tech. Rep. UUCS-93-013, Univ. of Utah, June 1993.
25. SRIRAM, D., AND LOGCHER, R. The MIT dice project. *IEEE Computer Magazine* (Jan. 1993), pp. 64–71.
26. TAKANO, M., MASAKI, H., AND SASAKI, K. Concept of total computer-aided design system of robot manipulators. In *Robotics Research: 3rd Int. Symp.* (1986), pp. 289–296.
27. WILL, P. Information technology and manufacturing. CSTB/NRC Preliminary Report 1, National Academy Press, Nov. 1993.