University of West Bohemia in Pilsen
Department of Computer Science and Engineering
Univerzitni 8
30614 Pilsen
Czech Republic

Redesign of CHAMP Control System Using SOFA Components

Full-length version of a conference paper

Sven-Arne Andréasson, Jan Valdman

Technical Report No. DCSE/TR-2001-06
November, 2001

Distribution: public

Technical Report No. DCSE/TR-2001-06
November 2001

Redesign of CHAMP Control System Using SOFA Components

Sven-Arne Andréasson, Jan Valdman

Abstract

This paper describes a component decomposition of a CHAMP control system for flexible manufacturing. A new active database approach is introduced to simplify relations and messages within the system. The system is described in terms of SOFA software components and takes advantages of SOFA s ability of formal description of component interfaces, relations and behavior provided by Component Description Language. Other existing features that can support flexible manufacturing like Dynamic Component Updating are also mentioned in the paper.

This work was supported by the Grant Agency of Czech Republic, project No. 201/99/0244 "Developping software components for distributed environment".

Copies of this report are available on
http://www.kiv.zcu.cz/publications/
or by surface mail on request sent to the following address:

University of West Bohemia in Pilsen
Department of Computer Science and Engineering
Univerzitni 8
30614 Pilsen
Czech Republic

REDESIGN OF CHAMP CONTROL SYSTEM USING SOFA COMPONENTS

Sven-Arne Andréasson, Jan Valdman
DCS, Chalmers University of Technology, andreasson@cs.chalmers.se
DCSE, University of West Bohemia, valdman@kiv.zcu.cz

This paper describes a component decomposition of a CHAMP control system for flexible manufacturing. A new "active database" approach is introduced to simplify relations and messages within the system. The system is described in terms of SOFA software components and takes advantages of SOFA's ability of formal description of component interfaces, relations and behavior provided by Component Description Language. Other existing features that can support flexible manufacturing like Dynamic Component Updating are also mentioned i the paper.

1. INTRODUCTION

To achieve a control system that is easy to adjust to different physical environments a good choice is to design it as a number of well defined components. In this paper the CHAMP reference model is described as components using the SOFA component software architecture. To minimize the complexity of the protocols among these components the concept of an active database is also used. Together this gives a control system design where the need for reprogramming when used for another physical environment is minimal.

1.1. CHAMP Overview

The CHAMP reference model was created to give a general control system that could be used for many types of manufacturing without reprogramming of the system (Adlemo et al, 1995). When used for a different system there should be need only for writing routines for adjusting to different new machinery, "drivers". Then the rest needed adjustments are made by inserting system information in the database (Gullander et al, 1998).
The reference model is built up with components as in Figure 1. Products are described using a separate program that gives a description that consists of operations and the order in which the operations must be performed. These descriptions are stored in the database. The database also contains information about all the resources and in which states they are. There are also mappings of which programs that can perform the operations on which resource. This information is used by the scheduler in order to decide what action to perform next by the system. Then the different states

of the resources are considered in order to only give schedules that can be performed at the present situation in the system (Fabian et al, 1997)

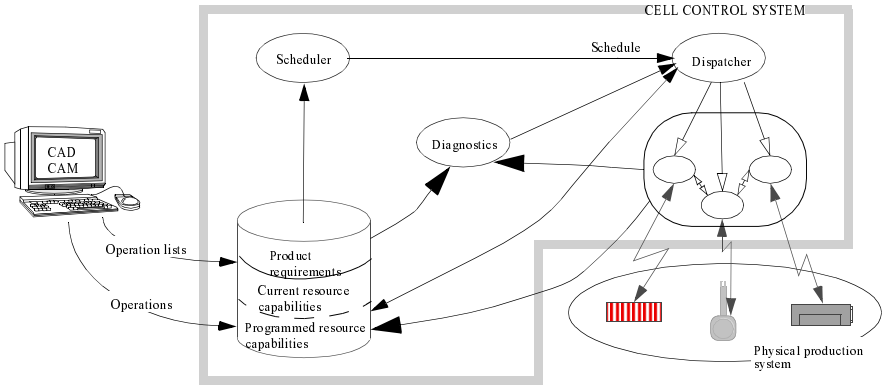
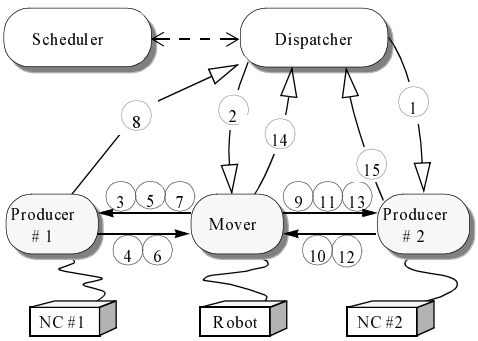


Figure 1 –The CHAMP model.

The schedules are given to a dispatcher process that sends out orders to the different resources. When designing the CHAMP model there has been an effort to minimize the number of messages sent between the dispatcher and resources, "vertical" messages, but instead use messages between the different resources, "horizontal" messages (see Figure 2) (Gullander et al, 1995). Whenever changing their states the resources update the database. Thus the scheduler always calculates the schedule according to the actual states. The dispatcher process also has an operator interface showing the production to the operator. The operator can intervene in the system by gradually taking over the dispatcher's role and thus decide about the schedule or in the most manual mode, send his own commands to the resources (Adlemo et al, 1997). There are also possibilities of adding other programs to the system, such as diagnosis and error recovery programs.

1.2. SOFA Overview

SOFA is a component software architecture created with an assumption that in the future software applications will be built of many small, independent and reusable modules. SOFA component is a black box of some type with precisely defined inter-



1: *PREPARE*
2: *MOVE_FROM_TO*
3 - 7: *Handshake messages*
8: *POST_HANDSHAKE_READY*
9 - 13: *Handshake messages*
14: *PRE_HANDSHAKE_READY*
15: *OPERATION_READY*

Figure 2 – The Resource - Dispatcher protocol

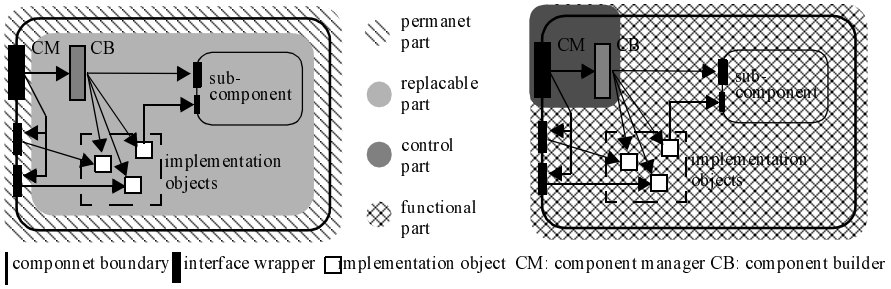


Figure 3 – Internal Structure of a SOFA Component

faces. A SOFA application is a hierarchy of mutually interconnected software components. An application is created just by a composition of bought components perhaps with great reuse of already owned components (Plasil et al, 1998).
Components are described in Component Description Language (CDL) that is a high level structured language capable to describe component structure (interfaces, frames, architecture, bindings) and behavior (protocols, state machines) (Plasil et al, 1999).
SOFA framework allows upgrading of components at runtime via Dynamic Component Updating (DCUP) features. At runtime, each component consist of permanent and replaceable part, as it is shown in Figure 3.

2. SOFA DECOMPOSITION OF CHAMP

Component decomposition of CHAMP with beneficial usage of active database can considerably simplify the control loop, easily describe relations between modules and thus allows more compact and clear control system. This is a third generation design of CHAMP/FMS, moving from an unstructured message-passing model, via modular TCP/IP-based client/server model towards a middleware component architecture.

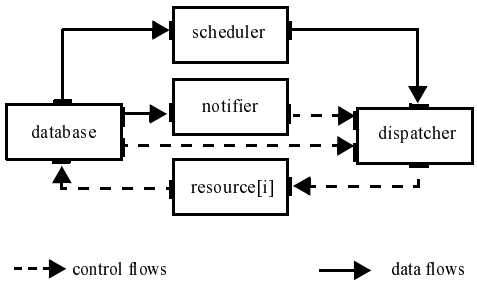


Figure 4 – SOFA Decomposition of

2.1. Concepts

The component decomposition provided in this paper is based on “active database” approach. It means that there is a possibility for a control loop where all control information is propagated only in one direction. The feedback information of the controlled system is provided only by updates of a central database. Compared to previous version of CHAMP (Andréasson, 2001), the “vertical” commands are reduced even further: all “upwards” commands are removed and replaced by the active database approach.
Some modification also reflect the benefits and limitations of SOFA architecture.

2.2. Components

This section provides an informal description of each component and explain in more detail its function. A formal SOFA description using CDL is provided in the next chapter.

2.2.1 Active database

The database contains the state of whole manufacturing cell and all information about products, resources, operations and programs. Active database is a composition of a regular relational database and a wrapper that monitors incoming updates and notifies subscribed components about the corresponding updates. Thus the dispatcher is notified about state changes.

Since regular relational databases are not active, we introduce a notifier component. This component monitors database updates and notifies other components that have subscribed for certain variable changes. Components can subscribe for changes of wanted variables. For example, the dispatcher can subscribe for change of "running state" variable of a given resource in order to recognize an end of moving operation etc. Also the operator interface can be updated; the point here is that both the dispatcher and the operator must have the same view of the cell.

2.2.2 Scheduler

Scheduler is responsible for giving the possible operations to be performed when asked by the dispatcher. It prevents possible deadlocks by removing unsafe operations. The scheduler might also optimize the production by only choosing certain operations.

At the beginning, the scheduler gets a list of present products in the cell. For each product it gets an operation list that is parsed to find out which operations to perform on the product and in which order. Scheduler also finds which resources are capable for each operation.

2.2.3 Dispatcher

The dispatcher is an executive body of the control system. It asks the scheduler for next possible commands and chooses one among them. Then the dispatcher sends out appropriate orders to the corresponding resources using a high-level "vertical" protocol. Then it asks the scheduler again for next possible command. The dispatcher also asks when notified that a new product has entered the cell or after a time-out.

The scheduler might send commands that can not be executed immediately. Then the dispatcher waits until the resources are available for one of the commands and that command is chosen.

There is also a "manual mode" when the operator chooses between the commands given by the scheduler.

2.2.4 Resources (Movers, Producers, In/Out Buffers)

The operations needed for the products are performed by resources. To do this there must be a NC program for the corresponding operation and resource. The products are moved by movers between resources. During cell operation, the resources update the database and it thus reflects inner states of the resources. Movers performs handshakes with corresponding resource when getting/leaving products. These handshakes follows separate "horizontal" protocols.

The resources have a main state, a running state and a handshake state. Main states are RUNNING, STOPPED and ABORTED and they indicate the overall state of the resource (see Figure 5).

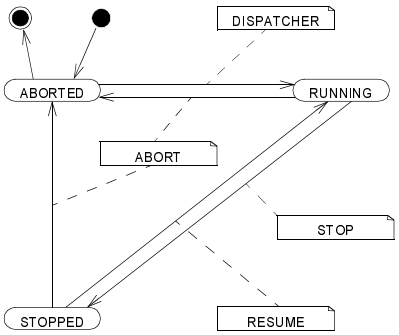


Figure 5 – The Main States for a Resource

When the main state is RUNNING, we can identify several running states that reflect the progress of production. They are IDLE, PREPARING, LOADING, LOADED, PROCESSING, FINISHED, UNLOADED and UNCERTAIN (see Figure 6).

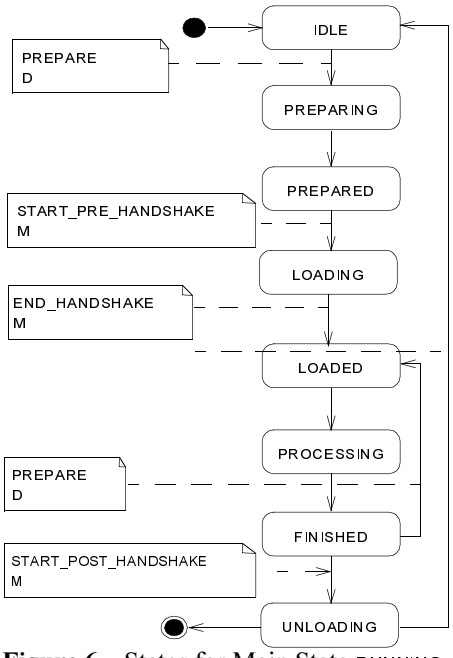


Figure 6 – States for Main State RUNNING

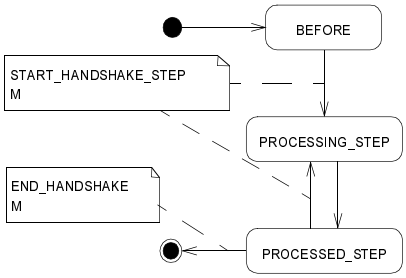


Figure 7 – Handshake states for a Resource

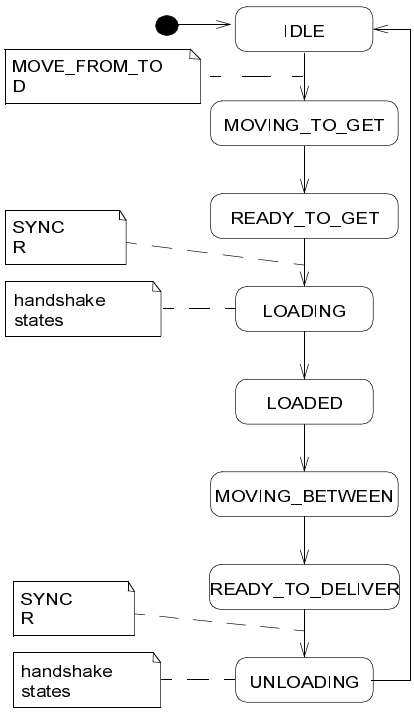


Figure 8 – The states for a Mover

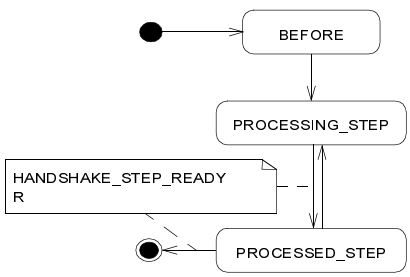


Figure 9 – Handshake states for a Mover

In running states LOADING and UNLOADING there occur resource-to-resource handshakes that can be described in terms of handshake states (see Figure 7).
A Mover can act as a Resource and subsequently have the same states and events as described above. However, when acting as a Mover it can also behave as the state machine described in Figure 8. Then its handshake states are described as in Figure 9.

2.2.5 Communication

Due to the database the communication is different then in Figure 2. Instead it will be like in Figure 10.
Let us look at a typical command:
At the beginning, the dispatcher sends a "prepare" command to resource P2 (see Figure 2). P2 responds by internal state changes that are visible in the database. Later, the dispatcher sends a "move from to" commands to mover M. M responds also by internal state changes, moves to P1, starts a handshake with P1 to get a products. During this stage, both P1 and M update their state changes in the database. Then M moves with the product to P2 and starts another handshake to leave the product; state updates of M and P2 occur.
All feedback is provided through the database. The dispatcher monitor state updates and mirrors the states of the resources. Although all state changes are recognized and shown on the operator's interface, only certain stage changes will cause the dispatcher to take new actions. In case of problems (time-outs etc.) it starts some error recovery actions.

2.2.6 Initial Cell setup

At the very beginning the database component is started because it acts as a server for all other objects (in SOFA terms, it has only provides-type interfaces). Then all resources in the cell log on the database and register their state. Each resource is represented by one SOFA component. Then the scheduler is started and it also logs on the database. Now the dispatcher can be started; it logs on the database, there it finds present resources (i.e. cell configuration) and logs on each resource. At last, the notifier component is instantiated; it has all interfaces of requires-type, so this must be the last instantiated component.

2.2.7 Limitations

The model of a control system provided here has some simplification to make the ideas simple. All issues mentioned bellow are theoretically solved, described in ear-

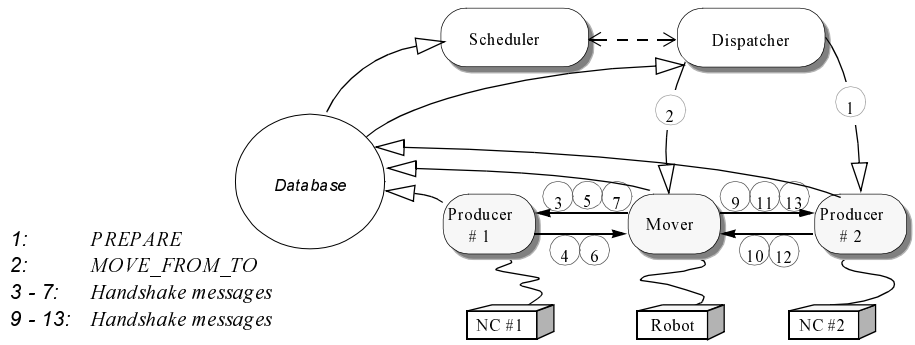


Figure 10 – The Resource - Dispatcher protocol with active database

lier papers and can be integrated into the model. These simplification originate both in SOFA and CHAMP and cause some limitations of the control system:
1. At present, there can be just one product in a resource at a time, so this version is not suitable for assembly cells. Needed extensions are not too difficult; it is only the resource implementation that has to be extended.
2. Resource to resource communication (i.e. handshake messages) can not be described using SOFA terms. Currently, SOFA allows only static component binding (described in CDL) and any temporary dynamic bindings are not supported yet.
3. To keep this SOFA/CHAMP model simple, we omit error recovery and other features that are essential for any realworld control system.

3. SOFA DESCRIPTION OF CHAMP

Here follows a formal description of the control system components using SOFA notation. SOFA describes modular systems in terms of interfaces and component architecture. A component type (in SOFA called a frame = abstract data type) is specified by its name, incoming and outgoing interfaces (i.e. by keywords provides and requires) and other optional features like behavior protocols or state machines.
As it was explained in the previous paragraph, we omit error recovery, diagnostic and operator interface that would make the description more complex and more difficult to read and understand.

3.1. Database Component

The database component is a passive SOFA component abstraction for a DBMS. Together with the notify component it forms the "active database" component. The database component provides interfaces required by the scheduler, the resources and the notifier component.

```
frame database{
 provides db4scheduler s,
          db4resources r[],
          db4notify n;
}
```

3.2. Scheduler Component

Since CHAMP is intended for experimenting with different scheduling algorithms, the SOFA description gives only the interfaces. For example, SOFA/DCUP allows to replace the scheduler with different versions during runtime and thus enables experiments with different scheduling algorithms.
In general, the scheduler get all its information only from the database using `db4scheduler` interface and then, when asked by the dispatcher via `scheduler4dispatcher` interface, it creates some scheduling information.

```
frame scheduler{
  requires database4scheduler db;
  provides scheduler4dispatcher d;
}
interface db4scheduler{
  product[] getProducts();
  operationList getOperations(produ
  getResources(operation);
}
interface scheduler4dispatcher {
  commandStructure getNext();
}
```

3.3. Resource Components

Resources are SOFA components representing the hardware components of the system. Resources talk to the dispatcher, to the database and to each other. All types of resources (including movers and buffers) share the same component frame but the usage of interfaces and state machines or behavioral protocols can differ.
A resource component uses (possibly multiple) state machine to check the correctness of incoming method calls. In SOFA, state machines or behavioral protocols can work in interface-scope (to check one interface call protocol) or in frame-scope (to check one component) or in architecture-scope (to check interaction of several components).

```
frame resource { // R/M/H StateUpdate after each transition !!!
 requires db4germ db;
 provides resource4dispatcher d,
          mover4dispatcher m,
          hadshakeInterface h;
 state machine running {    // Resource
    initial=idle;
    idle:       d.prepare -> preparing;
    preparing:  <internal> -> prepared;
    prepared:   h.start_pre_handshake -> loading;
    loading:    h.end_handshake -> loaded;
    loaded:     <internal> -> processing;
    processing: <internal> -> finished;
    finished:   h.start_post_handshake -> unloading;
    unloading:  h.end_handshake -> idle;
 }
 state machine resource_handshake { //hsh separated from resource
    initial = before;
    before:           h.start_handshake_step -> processing_step;
    processing_step:  <internal> -> processed_step;
    processed_step:   h.start_handshake_step -> processing_step;
                      h.end_handshake -> running.loaded;
 }
 state machine running{ // mover+handshake together
    initial=idle;
    idle:             d.moveFromTo->moving_to_get;
    moving_to_get:    <internal> -> ready_to_get;
    ready_to_get:     h.sync -> loading;
    loading:          <internal> -> before_loading;
    before_loading:   <internal> -> processing_step;
    processing_step:  <internal> -> processed_step;
    processed_step:   h.handshake_step_ready -> processing_step;
                      h.handshake_step_ready -> loaded;
    loaded:           <internal> -> moving_between;
    moving_between:   <internal> -> ready_to_deliver;
    ready_to_deliver: h.sync -> unloading;
    unloading:        <internal> -> before_unloading;
    before_unloading: <internal> -> processing_step_U;
    processing_step_U:<internal> -> processed_step_U;
    processed_step_U: h.handshake_step_ready -> processing_step_U;
    h.handshake_step_ready -> idle;
 }
```

```
}
 // main state methods:
 abort();
 stop();
 resume();
 // running state methods:
 prepare();
 // for recovery:
 startPreHandshake();
 startProcessing();
 startPostHandshake();
 setRunningState();
 getState();

 state machine {
    initial=running;
    final=aborted;
    running: prepare->running,
             stop->stopped,
             abort->aborted,
             startPreHandshake->running,
             startProcessing->running,
             startPostHandshake->running,
             resume->running;
    stopped: resume->running,
             setRunningState->stopped,
             stop->stopped,
             abort->aborted;
    }
}
```

```
interface db4resource {
 getResources();
 getResourceData(resource);
 setResourcePresent();
 updateRunningState();
 updateMainState();
 updateHandshakeState();
 state machine{...}
}

interface mover4dispatcher {
 moveFromTo();
 // for recovery:
 moveTo();// without product
 moveBetween();// with a produc
}
interface resource4diagnostic{
 whoAreYou();
 getVariables();
 getVariable();
 }
```

3.4. Dispatcher Component

The dispatcher is the executive manager of the system; it is an active component that makes client connections to all resources, and to the scheduler. For the control, the knowledge about resource's internal states have been minimized (see active database). However, since it provides human operator interface, it has to know all states that the operator is interested in.

```
frame dispatcher {
 requires scheduler4dispatcher s,
          resource4dispatcher r[],
          database4dispatcher d;
 provides dispatcher4notify n,
          operatoraccess o;

 state machine {
    initial=started;
    started:       d.getResources() -> no_work;
    no_work:       s.getNext -> try_work;
    try_work:      <internal> -> new_task;
                   n.runningStateUpdate -> no_work
                   n.runningStateUpdate -> try_wor
                   n.mainStateUpdate -> try_work,
                   <timeout> -> no_work,
                   n.newProduct -> no_work;
    new_task:      r[r2].prepare -> dest_prepared;
    dest_prepared: r[m].moveFromTo(r1,r2) -> no_wc
 }
}
```

3.5. Notifier Component

The notifier together with the database component forms an active database component. This component can be implemented by using trigger features of a DBMS or by polling the database. The notifier is used to supply the dispatcher with necessary changes in the system.

```
interface db4notify {
  getNextUpdate();
}
interface dispatcher4notify {
  runningStateUpdate(germ,state)
  mainStateUpdate(germ,state);
  newProduct();
}
```

As it was suggested before, it can use some trigger features of a DBMS or it can acts like an wrapper of a DMBMS: for example, it can monitor passing SQL commands and detect changes of certain tables, rows or attributes.

4. CONCLUSION

The paper provides a component-based description of the CHAMP control system using SOFA terms. The paper also introduces the “active database” approach that is used to simplify control structures of the original CHAMP model. Simultaneously we have also shown that we can get a good system decomposition using components.
Now, when whole control system is described in CDL, there is a comprehensive decomposition, clear component interfaces and clear component relations (bindings) and interaction. SOFA brings also the necessary formal level of CHAMP description. More benefits of SOFA decomposition originate in DCUP: advantages of multiple versions, multiple parallel implementations of each component and instant component upgrading/exchange are straightforward either for experiments and simulations during further CHAMP development or for realworld control systems.

5. REFERENCES

1. Adlemo A, Andréasson S-A, Fabian M, Gullander P, Lennartsson B. Towards a Truly Flexible Manufacturing System. Control Engineering Practice, vol. 3, no. 4, 1995, pp. 545-554.
2. Adlemo A, Andréasson S-A, Gullander P, Fabian M, Lennartsson B.Operator Control Activities in Flexible Manufacturing Systems. International Journal of Computer Integrated Manufacturing, vol. 10, no. 1, 1997, pp. 221-231.
3. Andréasson S-A, Brada P., Valdman J: Component-based Software Decomposition of Flexible Manufacturing Systems. In: Proceedings of ICCC’2000, Podbanske, 2000.
4. Andréasson S-A. Commands And States In The Champ Dispatcher. will appear in Proceedings of INCOM 2001, Vienna, 2001.
5. Fabian, M, Lennartsson B, Gullander P, Andréasson S-A, Adlemo A. Integrating Process Planning and Control of Flexible Production Systems, in ECC'97, Brussels, Belgium, July 1997.
6. Gullander P, Fabian M, Andréasson S-A, Lennartsson B, Adlemo A. Generic Resource Models and a Message-Passing Structure in an FMS Controller. Proceedings of the 1995 IEEE International Conference on Robotics and Automation, ICRA’95, Nagoya, Japan, May 1995, pp. 1447-1454.
7. Gullander P, Andréasson S-A, Adlemo A. Database Design for Flexible Manufacturing Cells. *Control Engineering Practice*, vol. 6, no. 11, November 1998, pp. 1411-1420.
8. Plasil F, Balek D, Janecek R: SOFA/DCUP Architecture for Component Trading and Dynamic Updating. In: Proceedings of ICCDS’98, Annapolis, IEEE CS, 1998.
9. Plasil F, Visnovsky S, Besta M: Behavior Protocols and Components. Tech. report No. 99/2, Dept. of SW Engineering, Charles University, Prague, 1999.