Proceedings of the 2001 IEEE/RSJ
International Conference on Intelligent Robots and Systems
Maui, Hawaii, USA, Oct. 29 – Nov. 03, 2001

DCA: A Distributed Control Architecture for Robotics

Lars Petersson[1] David Austin[2] Henrik Christensen[1]

[1]Center for Autonomous Systems,
Royal Institute of Technology,
Stockholm, Sweden.
`larsp, hic@nada.kth.se`

[2]Robotic Systems Lab,
Australian National University,
Canberra, Australia.
`d.austin@computer.org`

Abstract

Many control applications are by nature distributed, not only over different processes but also over several processors. Managing such a system with respect to the startup of processes, internal communications and state changes quickly becomes a very complex task. This paper presents a distributed control architecture which supports a formal model of computation as described by [1]. The architecture is primarily intended for robot control but has a wide range of potential applications. We motivate the design and implementation of the architecture by discussing the desired properties of a robot system capable of doing real-time tasks like manipulation. This leads to functionality such as a process algebra controlling the life-cycle of the processes, grouping and distribution of processes and internal communication transparent to location. Our implementation does not in itself introduce any bottlenecks due to a tree structure with local control over processes which gives an efficient and scalable architecture. At the end, an example scenario in which a fairly advanced problem like opening a door using a mobile robot with a manipulator arm is demonstrated in the presented framework.

1 Introduction

Robotics involves the development of complex, large control systems that must operate with a significant and highly varying bandwidth. Usually, software development takes place in groups and so it is important to note that robotic control systems will generally be modular in nature, consisting of components from a number of programmers. This paper presents the design and implementation of an architecture (named DCA) for control of distributed systems, which supports communications and modularity, and which is scalable to enterprise level applications.

There have been a number of prior efforts to develop an architecture suitable for (mobile) robotic systems. Unfortunately, none has yet demonstrated the potential to build large-scale, distributed systems for research and development for mobile robotics. For commercial applications a few systems such as ControlShell and Automation Ware have been developed but they are all proprietary and not available for a wide range of different platforms as often needed in R&D efforts. Simmons has developed and extended the Task Control Architecture (TCA) [2] over a number of years. TCA is used on a centralised process control model with a single supervisory module. For communication TCA uses either of the IPC or TCX packages based on a centralised communications server. From a process point of view the centralised communications poses a challenge for highly distributed systems. Also, the Distributed Architecture for Mobile Navigation (DAMN) [3], that in part is constructed on top of TCA components, utilizes a centralised server. In addition coordination is carried out by a centralised scheduling/arbitration mechanism which is a strong constraint in a general system. The Saphira architecture used by ActivMedia robots was developed by Konolige [4]. Saphira has as its core a common representation named Local Perceptual Space (LPS) that is a type of shared memory or blackboard for interprocess communication/coordination. LPS provides a convenient mechanism for communication but at the same time it introduces a particular model of computation and control that restricts highly distributed systems due to bandwidth limitations. The Intelligent Service Robot (ISR) architecture previously developed at the Center for Autonomous Systems, Sweden [5] was designed specifically for robotic applications, built on the Adaptive Communications Environment (ACE) [6] package. This architecture included a centralised supervisor and its limitations clearly demonstrated the need for distributed mechanisms for coordination.

It is also interesting to compare to other common distributed systems in existence today. For example, the Internet Chat Relay Protocol (IRC) [7] is a distributed "instant" messaging application used to transmit messages rapidly all over the world. However, the current implementation of IRC is not scalable, it has a requirement that all servers know about all other servers [7]. Work is underway to address this problem but this example illustrates the difficulties of creating a truly decentralised system, even when the requirement is obvious.

For process communication a number of new efforts have been developed. A protocol worth consideration is CORBA [8], which provides portable, open-standard communications. However, today there are only limited packages available that support truly real-time coordination. RT-CORBA, which is a real-time extension to CORBA may be a good alternative in the future. Another mechanism that is widely used is COM, which unfortunately primarily is used on the Microsoft Platform. Use of COM thus has limited use in a multi-platform environment. Another communications package, the Adaptive Communications Environment (ACE) [6] also provides for portable communications. The package uses many advanced templates which requires use of the latest C++ compilers and limits its availability for use on hard real-time platforms.

The lack of satisfactory packages for the development of robotic software systems has lead us to develop a new system that addresses the limitations of current packages. The new system is called DCA. Section 2 presents the requirements for a distributed control architecture and also describes the design of the language that has been implemented for our DCA. In Section 3 we describe the Modular Software Development Environment (MSDE) which has been developed to provide portable communications and process management for real-time systems. Next, Section 4 describes the implementation of DCA and Section 5 presents an example of a real-time system described in the framework of DCA.

2 Design Overview

The DCA system was designed with a mobile robot performing manipulation in mind. In such an application many critical issues arise and it is therefore appropriate to discuss the requirements for a distributed control architecture first.

- **Modularity**: An important consideration when designing any large, complex system is to break it into pieces for development and testing. This enables incremental progress of the development and large groups to work together on the same system. Experiences from earlier work show that it is often a considerable amount of work to re-use code in a new system. In a truly modular system where there is no need to modify basic functionality, a library of control modules can be built.

  Solution: The motivation for the development of DCA was to provide support for modular systems and to allow teams of programmers to cooperate on large systems. The language is modular in nature and much effort has been spent to ensure that the implementation of new modules is as simple as possible.

- **Scalability**: A robotic system that will be solving anything more advanced than toy problems require a structure that does not suffer from scalability problems. In fact, this is one of the more important requirements of any robot control system. Our system requires about 10-50 different processes to do mapping, navigation, object recognition and object manipulation. The communication and control of these easily become a bottleneck limiting the expansion of the robotic system. The scalability requirement has many far-reaching consequences and must be kept in mind at all stages of the design and implementation. Many existing architectures do not scale well because of a central bottleneck, such as a centralised supervisor or a blackboard concept where a shared memory segment is used by all processes.

  Solution: DCA has been designed with scalability in mind. The DCA language (described below) permits hierarchical constructions and, using a hierarchical design, users can implement systems with no centralised portions which could be bottlenecks.

- **Efficiency**: While computers are increasing in speed very rapidly, it is still important to ensure that a control architecture is efficient so that high frequency control tasks can be undertaken. However, if there is a conflict with the next point, i.e. flexibility and generality, it is in many cases better to sacrifice efficiency over flexibility.

  Solution: The approach here was to have distributed event based decisions so that e.g. a real-time task can be separated from non real-time tasks if necessary. In addition, a low overhead protocol is used in the communication.

- **Flexibility and Generality**: The architecture for a distributed control system should be as flexible and general as possible without imposing any fixed structure. In research, it is desirable to try out new ideas in a simple manner rather than having optimal efficiency.

  Solution: The communication options in DCA allows the user to choose peer-to-peer, many-to-one, or one-to-many communications. Also, pushing or pulling of data can be decided by the user. Processes can be executed on different hosts and are easily shifted around if necessary.

- **Theoretical foundation**: The control of the system should be made using a theoretically sound foundation that allows synthesis and verification. Robotic systems become increasingly complex and it is therefore necessary to be able to do verification of a system that has been designed. This is especially an issue if the control system is going to work in a public setting where a failure in the control may cause fatal accidents.

  Solution: This was addressed by using a process algebra adopted from a formal model of computation which is described in detail in [1]. This algebra provides the potential of direct task-level specification in a manner which is human friendly as well as suitable for automatic planners.

2.1 Language design

The language that describes the control system was designed to allow abstraction of functionality, i.e. if a group of processes solves a well defined task, that group can be given a name and has a set of inputs and outputs. In fact, that group is in itself treated as a single process from an external viewer. In the following text it is assumed that a "process" can be either a single process or a group of processes.

Defining a process group requires the following entries:

- **Process type and arguments**: The process group is given a name reflecting the functionality and a variable number of symbolic arguments. These arguments are set by any other process group that instantiates this process.

- **Host**: The host on which the supervisor of this process will run is specified. The supervisor is a process controlling the internal execution of this process and it will be explained in detail in Section 4.

- **List of instantiated processes**: A process group must instantiate the processes it will use. If they have arguments, they have to be set here, either symbolically or as constants. There can be several instances of the same process type. A common use of the arguments is to pass the hostname on which it should run.

- **External I/O**: If the process group wants to export certain inputs or outputs they should be declared here. An external I/O is actually an internal I/O exported under another name.

- **List of internal connections**: Every instance of a process is, if connected, associated with one or more sets of internal connections. Every set of connections is given a label which can be referred to in the next entry. The possibility to assign several sets of connections to the same instantiated process makes it possible to change connections during the execution of the process algebra. The convention used is to connect outputs to inputs.

- **Internal event actions**: This is the section where the process algebra mentioned earlier is used. By using a set of operators, the execution of the instantiated processes in this group is controlled. The operators and an example are described in Section 2.1.1.

A typical example of a process is shown below. It acts as a compliant motion controller for a puma arm. When the process "hit" detects high forces it preempts the whole group with the use of the preemption operator "#".

```
process CompliantMotion(host1, host2){
   host{host1}
   instantiated_processes{
      forcedata instof ForceSensor(host1);
      compliant instof CompliantCtrl(host1);
      puma instof Puma560(host2);
      hit instof HitDetector(host1);
   }
   external_io{
      // This process has no external I/O
   }
   internal_connections(forcedata, lbl1){
      forcedata, Out1 -> compliant, In1;
      forcedata, Out1 -> hit, In1;
   }
   internal_connections(compliant, lbl1){
      compliant, Out1 -> puma, In1;
   }
   internal_event_actions{
      // Perform compliant motion until the process
      // 'hit' detects abnormal forces and preempts
      // the running group of processes

      hit # (forcedata[lbl1], compliant[lbl1], puma)
   }
}
```

2.1.1 The process algebra

Processes can generate different kinds of events at run-time that need to be taken care of. The events are e.g. DONE, ABORTED or error events like SIGSEGV or similar. The paradigm used is a completely event driven life-cycle of the instantiated processes. A suitable model for this has been developed in [1] where a number of operators are defined. These operators are:

- Concurrent Composition: $T = (P, Q)$. The process T behaves like P and Q running in parallel.
- Sequential Composition: $T = P; Q$. The process T behaves like P until that terminates and then behaves like process Q.
- Conditional Composition: $T = P^v : Q_v$. The process T behaves like process P until that terminates. If P aborts, then T aborts. If P terminates normally, then the value v calculated by P is used to initialize the process Q, and then T behaves like Q.
- Disabling Composition: $T = P \# Q$. The process T behaves like the concurrent composition of P and Q until either terminates, then the other is aborted and T terminates.
- Synchronous Recurrent Composition: $T = P^v :; Q_v$. This operator is recursively defined as $P^v :; Q_v = P^v : Q_v; P^v :; Q_v$. If P terminates normally, its calculated value v will initialize Q (as in the conditional ':' operator). Then when Q terminates, the expression becomes the same as the initial expression. However, if P aborts, the whole expression is aborted.

The value v passed between the processes can be of any size, it is the responsibility of the receiving process to interpret the data.

The operators above can e.g. be used in an opportunistic planning mechanism. Consider an example where the robot is tidying up a room where the objects are randomly spread out. The robot knows where every object is supposed to be, but to solve the task, no particular order in which objects are put in place is implied. At the same time, we do not want the robot to spend infinitely much time searching for hidden objects so there must be a timeout as well. A simplified task description of this scenario could look like this:

```
timeout # (FindObjectA:PlaceA, FindObjectB:PlaceB,
           FindObjectC:PlaceC)
```

where the three "FindObject" processes run concurrently and when either one of them terminates, the corresponding "Place" process will run. The whole expression will not terminate until either the timeout terminates or the objects A, B and C have been placed.

Of course, in a more realistic implementation of the above, there also have to be processes monitoring the availability of exclusive resources and preventing a timeout from happening while placing an object that is found.

3 Modular Software Development Environment

The Modular Software Development Environment (MSDE) provides support for the development of modular software systems. The heart of MSDE is a communications library called GeneralComms and a number of other services are used to support modular programming.

3.1 GeneralComms

GeneralComms provides an abstraction of communication and event-based programming across a number of platforms. GeneralComms was inspired by the Adaptive Communications Environment (ACE) package[6]. However, the ACE package requires a modern C++ compiler and results in very large compiled programs. In addition to event-based communications, GeneralComms includes methods for packing and unpacking messages in a portable manner.

3.2 NameServer

The NameServer allows processes to register their communication parameters (e.g. host and port) so that other processes can locate them by name in a portable fashion. The NameServer includes the ability to locate peer NameServers and to share information with them.

3.3 TimerServer

The TimerServer provides high-resolution timing events. Generally, hardware systems provide a limited number of high-resolution timers (often one!) and so the TimerServer permits user modules to share the hardware timer(s). The TimerServer does very little processing and so has a low overhead.

3.4 Executer

The Executer allows authenticated user programs to execute further programs on the same host. Each host which will be used within the distributed control system must have an Executer running. The Executer allows the DCA to start the components of the user's control system.

3.5 DBManager

The database manager provides access to a repository which can be used for configuration data, passwords and other data. At present, there is only support for a single, centralised database but we plan to implement decentralised databases. Decentralised databases are particularly important for mobile robotics. Consider a team of mobile robots moving around. To continue operation, each must have access to the database, however there are usually limitations on communications range. Therefore, we intend to replicate the database on each of the robots and use the concept of transactions to keep all of the databases updated as well as is possible in the face of an intermittent communications link.

4 Distributed Control Architecture

The distributed control architecture builds on the MSDE components described in Section 3 above and provides a suitable user interface.

The DCA consists of instances of two functionally different parts. There is a *supervisor*, DCAS, that organises the execution of a subset of *controller modules.*

4.1 The Supervisor (DCAS)

The supervisor has two means of receiving information about the processes to run, a parser reading a text file describing the control system or a binary communications interface. When invoking a control system, a DCAS is started with command line parameters specifying a text file. This is then interpreted, causing the DCAS to start the local controller modules it is in charge of as well as starting sub-DCAS supervisors if necessary. The information that is not used at the level of the first DCAS is then passed down to the appropriate DCAS on a sub level using the binary interface (see Figure 1). This is then performed recursively until the whole tree structure is started. It is here important to note that even if a process is "started" it is not running. To deal with real-time issues, all processes are in fact started beforehand, but inactive. When all processes are started, another walk through the tree is made to connect all the different modules. Also here, all connections are made beforehand but may be inactive. Connections between different process groups, i.e. managed by different DCAS, are resolved to go directly from one controller to another without passing any communication traffic through the DCAS. That makes the communication between process groups as efficient as within one group.

Further on, it is only when the root DCAS has received a confirmation that the whole tree is successfully configured that it gives the signal to run the system. Every DCAS is then only taking care of the local events, and only if the process group emits an event, it is talking to the DCAS on the level above.

Whenever a DCAS receives an event it is sent to the process algebra interpreter in the DCAS. It is from the impact of that event that the DCAS then decides which processes to initialise, run, stop or reconnect.

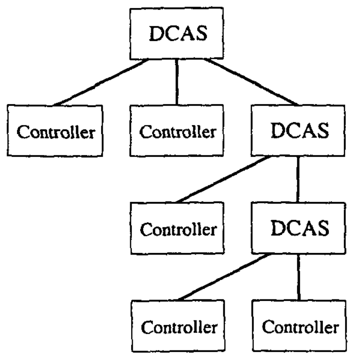

Figure 1: A typical tree built from a definition file read by the top DCAS. All subsequent DCAS receives their information from above. Connections between controllers are not shown in this figure. Note that every node in the tree can run on an arbitrary host.

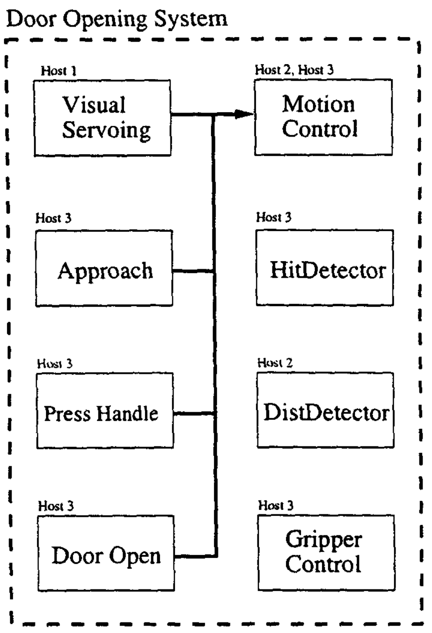

Figure 2: A complete door opening system. In this example, host 1 is dedicated to computer vision, host 2 controls the mobile platform and host 3 is running a real-time OS for manipulation tasks. Note that in the case of a compound process, there may be several hosts.

4.2 Controller Modules

The actual work is made by the controller modules which can be described as leaves in the hierarchy of processes. These have well defined communication interfaces with functions such as Init(), Run() and Stop(). The core functionality is located in a base class, so a user that wants to implement a new class of controller, only has to inherit from that and implement the controller specific parts. Typically, a user has to do these few steps:

- Assign names to inputs and outputs in a constructor.
- Write code to initialise sensors, hardware or local variables in the Init() function.
- Start a timer in the Run() function if the controller is repetitive.
- Write the control loop in a function MainCalc(). The inputs can be read, necessary calculation is performed and the result is sent to the corresponding outputs.
- Write code to gracefully disconnect from hardware and stop the timer in the Stop() function so that Init() can be called again.

5 Example Scenario

In this section a fairly advanced problem such as opening a door is considered. This task demonstrates a distributed problem, not only over several processes but also over a number of hosts. The process algebra proves useful in sequencing the actions although there is not any opportunistic planning as in the previous example. This example demonstrates that an otherwise very tedious task of distributing the processes over several computers and making them talk to each other is feasible in DCA. All the charachteristics of the system, including the state changes, is captured in one place.

Figure 2 shows a complete door opening system capable of locating the door handle by the use of visual servoing, detecting distance to the door with a laser range sensor, grasping and pressing the door handle and finally opening the door. The contents of compound processes are shown in the Figures 3 - 6.

All processes to the left in Figure 2 deliver velocity control signals to the motion controller process, which in turn has a redundancy resolution module inside as shown in Figure 3. In the case of a mobile manipulator, or any redundant system, it is necessary to distribute the motion in a suitable way over the degrees of freedom available.

Figure 4 shows the most complex process group where three different control schemes are fused

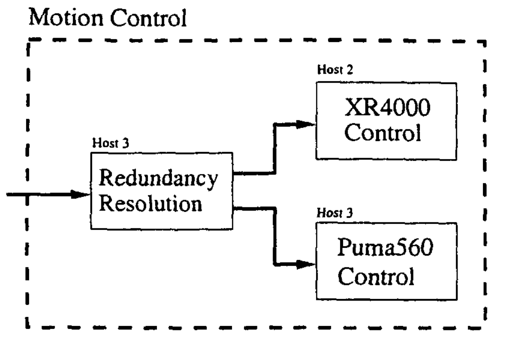

Figure 3: A process group taking care of redundancy resolution and control of the XR4000 and the Puma560.

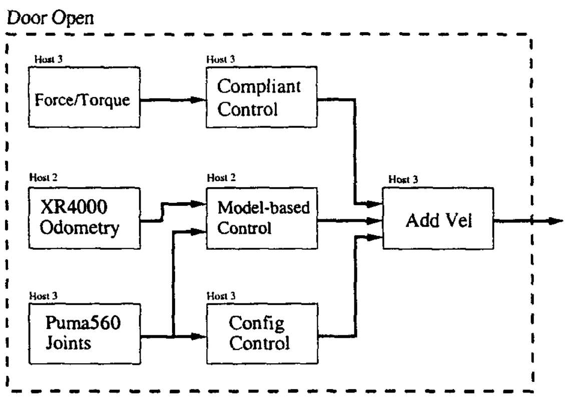

Figure 4: The process group solving the task of opening the door. Three different control schemes are fused into one velocity signal.

into one velocity control signal. This algorithm, although not in this framework, is described in the paper [9] and it proved successful in spite of its simplicity.

There are two processes monitoring special conditions (see Figure 5 and 6), the distance stopping criteria and the hit detector. When these processes terminate they are used to disable the concurrent processes.

A description of the task using the operators described earlier can look like this:

```
MotionControl , ((VisualServoing # DistDetector) ;
                 ((OpenGripper, Approach) # HitDetector) ;
                 CloseGripper ; PressHandle ; DoorOpen)
```

This states that the motion controller should run at all times, while the visual servoing continues until a certain distance has been reached and the distance detector disables it. Then, the gripper is opened while approaching the door handle. When a hit is detected, i.e. the gripper has reached the handle, that group is disabled and a sequence of closing the gripper, pressing the handle and opening the door is started.

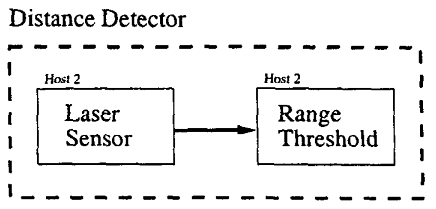

Figure 5: A process group monitoring the distance to the door.

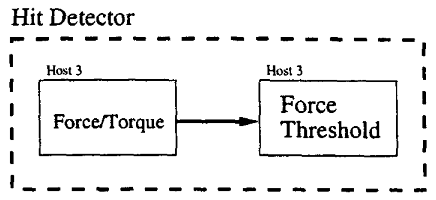

Figure 6: A process group monitoring the forces on the force/torque sensor.

An almost complete description of the processes is shown in the Appendix. Note that due to space limitations the internal connections are left out.

6 Conclusion

In this paper, we have presented an architecture for building distributed control systems. DCA has been specifically designed and implemented for modularity and scalability. In addition, we have used a process algebra for DCA, providing a theoretically sound basis and permitting verification. When designing and implementing DCA we have sought to maintain flexibility and generality whilst achieving efficiency. The result is very flexible and can achieve quite good real time performance. The example scenario presented illustrates the ease with which modular systems can be designed. In the absence of any convincing alternatives, we believe that DCA is a highly suitable tool for the implementation of distributed control systems.

Acknowledgements

This work was sponsored in part by the Swedish Foundation for Strategic Research through its Centre for Autonomous Systems at KTH.

References

[1] D. Lyons and M. Arbib, "A formal model of computation for sensory-based robotics," no. 3, pp. 280–293, 1989.

[2] R. Simmons, R. Goodwin, C. Fedor, and J. Basista, "Task control architecture programmer's guide to version 8.0." http://www.cs.cmu.edu/~reids, May 1997.

[3] J. K. Rosenblatt, *DAMN: A Distributed Architecture for Mobile Navigation.* PhD thesis, The Robotics Institute, Carnegie Mellon University, 1997. CMU-RI-TR-97-01.

[4] K. Konolige and K. Myers, "The saphira architecture for autonomous mobile robots." Book chapter, available at http://www.ai.sri.com/~konolige/sapphira/.

[5] M. Lindström, A. Orebæck, and H. Christensen, "A flexible architecture for a service robot," in *IEEE Conference on Robotics and Automation*, (Detroit, MI), p. (Submitted), May 1999.

[6] D. C. Schmidt, "Adaptive communications environment (ace)." http://www.cs.wustl.edu/~schmidt/ACE.html.

[7] C. Kalt, "Rfc 2810: Irc architecture." ftp://ftp.irc.org/irc/docs/rfc2810.txt.

[8] T. O. M. Group, "Corba/iiop 2.4 specification." See http://www.omg.org/.

[9] L. Petersson, D. J. Austin, and D. Kragic, "High-level control of a mobile manipulator for door opening," in *Proc. of Int. Conference on Intelligent Robots and Systems (IROS)*, 2000.

Appendix A

Realization of Door Opening

A realization of the door opening system in the framework of DCA is shown below. Note that internal connections are omitted due to space limitations.

```
process MotionControl(host1, host2){
   host{host1}
   instantiated_processes{
      xr4000 instof XR4000Ctrl(host2);
      puma instof Puma560Ctrl(host1);
      redundancy instof RedundancyResolution(host1);
   }
   external_io{
      In = redundancy, In;
   }
   internal_event_actions{
      // Run all processes concurrently
      redundancy[lbl1], xr4000, puma
   }
}

process HitDetector(host1){
   host{host1}
   instantiated_processes{
      force instof ForceTorque(host1);
      threshold instof ForceThreshold(host1);
   }
   external_io{
      // No external I/O
   }
   internal_event_actions{
      // Run both processes concurrently
      force[lbl1], threshold
   }
}


process DistanceDetector(host1){
   host{host1}
   instantiated_processes{
      laser instof LaserSensor(host1);
      threshold instof DistanceThreshold(host1);
   }
   external_io{
      // No external I/O
   }
   internal_event_actions{
      // Run both processes concurrently
      laser[lbl1], threshold
   }
}

process DoorOpen(host1, host2){
   host{host1}
   instantiated_processes{
      force instof ForceTorque(host1);
      compliant instof CompliantCtrl(host1);
      xr4000 instof XR4000Odometry(host2);
      puma instof Puma560Joints(host1);
      model instof ModelBasedCtrl(host1);
      config instof ConfigCtrl(host1);
      add instof AddVelocity(host1);
   }
   external_io{
      Out = add, Out;
   }
   internal_event_actions{
      // Run all processes concurrently
      force[lbl1], xr4000[lbl1], puma[lbl1],
      compliant[lbl1], model[lbl1], config[lbl1], add
   }
}

process DoorOpenSystem(host1, host2, host3){
   host{host3}
   instantiated_processes{
      visualserv instof VisualServoing(host1);
      approach instof LinearTrajectory(host3);
      presshandle instof PressHandle(host3);
      dooropen instof DoorOpen(host3, host2);
      motionctrl instof MotionControl(host3, host2);
      hit instof HitDetector(host3);
      distdetect instof DistDetector(host2);
      opengrip instof GripperControl(host3, "open");
      closegrip instof GripperControl(host3, "close");
   }
   external_io{
      // No external I/O
   }
   internal_event_actions{
      MotionControl ,
      ((VisualServoing[lbl1] # DistDetector) ;
       ((OpenGripper, Approach[lbl1]) # HitDetector) ;
       CloseGripper ; PressHandle[lbl1] ; DoorOpen[lbl1])
   }
}

process main(){
   instantiated_processes{
      dooropensyst instof DoorOpenSystem("c1.nada.kth.se",
                                         "c2.nada.kth.se",
                                         "c3.nada.kth.se");
   }
   internal_event_actions{
      dooropensyst
   }
}
```