The H Metalanguage and the H tools

Juan-Antonio Fernández-Madrigal, Cipriano Galindo Andrades, Ana Cruz Martín, Javier González

Technical report (Draft), May 2007
System Engineering and Automation Dpt., University of Málaga (Spain)

Abstract.
This technical report (draft) describes in detail the H metalanguage and the H tools for programming heterogeneous applications with distributed, concurrent, real-time, and fault-tolerance requirements. This is an evolution of our BABEL development system, which in turn was an evolution of our NEXUS programming framework.

Introduction

The integration of heterogeneous software applications, specially when they involve hardware programming, may be obstructed due to heterogeneity. This is specially important in areas such as CIM or robotics, but also appears in other programming environments (for example, the Web). Heterogeneity appears in different forms. To begin with, a wide variety of equipment can be found in the shop level of a factory: programmable controllers, robots, AGVs, sensors and actuators, CNC machinery, warehouses and material handling systems, and so on. Communications may also involve different kinds of networks with different requirements. Most of the hardware elements in a modern application must be programmed, either in a commercial-off-the-shelf (COTS) solution or in a more general purpose language (C, C++, Java…), which adds even more heterogeneity. Moreover, the programming skills of the staff involved in the development of an application will change depending on the complexity of their tasks.

In addition to the heterogeneity produced by the diversity of components needed, the adaptation to a changing global market unavoidably demands flexibility. Changes will affect to the wide range of heterogeneous elements enumerated previously, thus, in order to keep a manageable system, they must be treated in a simple and efficient fashion.

We claim that software is a good starting point to cope with heterogeneity. Programming languages are very close to the root of the problem and the plasticity of software makes it well situated to address it. For example, a common programming paradigm that could be used for all the equipment would reduce heterogeneity and increase flexibility. However, approaching heterogeneity by forcing the use of a single software paradigm or language is difficult to put in practice due to the need of all the manufacturers to adhere to it. It seems more appropriate to look for a (maybe simpler) solution that is able to integrate different programming approaches. This would produce better results than a common paradigm when confronted to very different needs and situations.

In this paper we follow this line of research and propose a solution to heterogeneity and flexibility from the language programming perspective, aimed to adapt as best as possible to the integration of heterogeneous components and to facilitate reusability and extensibility, and thus, to address appropriately the flexibility issue. Our approach is an evolution of a previous work on the same problem [<ref>], with a completely new formalization and features for fault-tolerance and communication frameworks. It consists of a simple and well defined metalanguage, called H, for designing and implementing distributed software modules (for example, programs in charge of controlling mechanisms) that can be connected through different classes of networks. Each of these modules, in turn, can complete its functionality with the writing of different programming languages (COTS, general purpose, etc.) in order to adapt to each particular device. The H metalanguage includes concurrence, fault-tolerance and real-time features to specify the different requirements of the application and the devices that compose it.

Furthermore, we have provided H with a set of tools that deal with heterogeneity from a wide perspective that covers all the stages of the development lifecycle of the application. In general, H and the H-tools enable two valuable features: extensibility and validation/debugging mechanisms. Extensibility allows the integration of any other new elements that are not considered in the proposed specification; this characteristic maintains both heterogeneity and flexibility, while reduces the software development costs. Validation offers the chance to check the proper function of the system prior to its deployment, while debugging is intended to obtain conclusions about execution performance of the system.

We have tested preliminary versions of the H and the H-tools (BABEL and NEXUS) in our automation research labs on robotic applications during the last ten years [<ref>]. In this paper we illustrate the new formalization with a case of study of a CIM application. The obtained results have been promising in both cases, having reduced the cost of development substantially by increasing flexibility while coping with heterogeneous devices with reduced effort.

The plan of the paper is as follows. In the next section we introduce the precise definition of the H metalanguage and describe the H tools. The following

section presents a case of study. Finally, we end with the conclusions of the presented work, along with an outline of some future work in this area.

The H Metalanguage and the H Tools

In this section we describe our specification language for the design and programming of heterogeneous applications, called H (for "Heterogeneity"), and the set of tools that produce, deploy, validate, and debug executable programs from those designs, This is a formalization and extension (mainly with multiple communication platforms and fault-tolerance capabilities) of a previous work called BABEL [<ref>]. H has been conceived for producing heterogeneous codifications (even in different programming languages) of distributed software with real-time and fault-tolerance requirements. Since it is essentially a description language, a specification written in H can be easily transformed into modern markup languages such as XML [<ref>], which provide good structure capabilities and a widely available set of tools for edition and validation.

H has the following main features:

a) *Heterogeneity enabled.* At all the levels of the design of the application (from the structural specification of programs to the deployment of the executable code), H allows the designer to use very different components, both hardware and software, in an integration framework that can take advantage of the best qualities of each one. Heterogeneity is properly encapsulated and managed, and the parts of the design that do not depend on concrete components are conveniently identified and isolated so they can be reused to the maximum.

b) *Distributed.* H is aimed to implement distributed applications over heterogeneous networks, possibly with fault-tolerance and real-time requirements. Of course, applications that run entirely on a single machine are also possible.

c) *Concurrent*. The main execution unit of an application specified with H is the module, which runs on a given computing platform (computer, PLC, embedded processor, etc.). Within the module, H enables the designer to specify concurrent execution threads, and can regulate concurrency at a fine granularity level by setting appropriate characteristics for the services of the module. Also, it provides mechanisms for synchronization and internal communication of threads.

d) *Real-time*. The design of the application can include real-time requirements in the execution time and in the communication between modules. A relative, hierarchical priority scheme is provided that can be mapped both to non-(or soft) real-time operating systems' facilities and to hard real-time platforms.

e) *Fault-tolerant.* The modules of an application have separated memory spaces and execution processes, so a fault in one should affect the minimum to others and to the integrity of the whole application. In addition, modules can be deployed with several replicas of each one. Mechanisms for supporting a variety of policies for distributing the work between the replicas and merging their computation results are provided, including both passive and active replication modalities [<ref>].

f) *Extensible*. It is easy to enrich H with any number of components not covered at this time. Also, H has some object-oriented characteristics, such as inheritance, which is available for reducing the cost and time of designing modules, and for improving reusability. Future work is to be done on including other object-oriented features such as polymorphism.

g) *Validation/debugging enabled*. All the information that can be retrieved from the application for validation is available from its H specification, thus different validation algorithms can be run on it prior to execution. Also, the application can produce logging results of its execution for off-line analysis.

In H, an application is composed of a set of modules (possibly distributed in a network), being each module executed on a single computational machine and providing a number of public services to other modules. During the design with H, the components of the application and its requirements are separated conveniently into the heterogeneous components and the portable ones. In the rest of this section we detail the different levels of the H metalanguage, its syntax, and its typical use for designing and implementing a CIM application.

The Portable Components

For improving the most the reusing of the work done in a given application, and for enabling heterogeneity at the design level, the parts of the design that are not tightly linked to specific hardware/software components (that is, the *portable* parts), are specified separately from the rest. The main portable

Figure <>

```
Module [abstract] structural design <module name> [inherits from /list_of_ids/ ]

   Description:  "Description of the module"

   Author: "Author name"

   [ Data definitions
          /list_of_HDL_defs/
     End data definitions ]

   [ Signal definitions
          /list_of_signals_defs/
     End signal definitions ]

   ( Service <service name>
          [ Characteristics: /list_of_serv_chars/ ]
          Priority: /service_prio_level/
          [ Inputs: /list_of_HDL_data/ ]
          [ Outputs: /list_of_HDL_data/ ]
          Description: "description of the service"
     End service <service name> )+

   { Event handler <signal id>
          Priority: /handler_prio_level/
          Description: "description of the handler"
     End event handler <signal id> }

   { Notification handler <signal name>
          Priority: /handler_prio_level/
          Description: "description of the handler"
     End notification handler <signal name> }

End module structural design <module name>
```

components in an H application are the modules. They are Active Objects [<ref>] which contain an internal status and provide some services and data definitions to other modules. In H, modules are specified at two separated levels: their structural design and their codification design. The latter can include weak references to non-portable components.

Structural Design. The structural design of a module covers its public definitions and services, and some characteristics that do not depend on any particular software/hardware. Thus, the structural design is completely portable and can be reused as much as possible. Each structural design must follow the syntax shown in fig. <>.

For improving the reuse of the structural designs, H includes a multiple inheritance mechanism (the **`inherits from`** keyword) that allows the designer to easily create modifications of existing designs, to combine several designs into a more complex one, and to make schemes of designs not intended for implementation (through the use of the **`abstract`** keyword). If a structural design inherits from another one, its definitions are always added to the parent's, except when they collide: in that case they replace them completely.

Within a module's structural design, services are public functions offered to other modules (and to the module itself). A service is the minimal sequential execution entity within a module. It can access the internal status of its module as described in the codification of modules, further on. It also can request other services and thus generate communications.

Services are of three types in H: *regular*, *event handlers*, and *notification handlers*. A regular service is requested by others or started by its own module when the module initiates (for example, a service for making a "homing" in a manipulator arm). An event handler service is executed only when an asynchronous event is sent by another service (for example, an alarm). Finally, a notification handler service is an event handler that only is visible within the module. All the services of a module can run concurrently within their module, but some of them can be set for blocking any other's execution. Table <> shows the characteristics that can be used for achieving different behaviors.

Each service in a module can have a priority of execution relative to the priorities of other services of its own module. This is part of the hierarchical relative priority system of H. At run-time, the priority assigned to each module (relative to the other modules' priorities) serves as the base for the priorities of its services. Services within a module are divided into four priority categories: *high (prioritized)*, *dynamic*, *numeric*, and *low (unprioritized)*. High priority services can pre-empt the execution of any non-high services of the same module. This is useful, for example, for assuring that a hardware monitoring algorithm never loses its time requirements. Low priority services can be pre-empted by any other service of the module. For

Table <>

Characteristic	*Effect*	*Typical use*
reentrant	A reentrant service can start its execution concurrently more than once and with other reentrant services, without waiting for the termination of other requests. It is not needed in **monitor** services or event/notification handlers, since they are always reentrant (and cannot be set as non-reentrant). In contrast, a non-reentrant service blocks the execution of any other service in the module, except monitors and event/notification handlers. Reentrant services require an appropriate use of shared resources, specially accesses to the internal status of the module. H provides suitable synchronization constructs for that purpose (see section <>).	-For improving the throughput of the module when more than one execution thread exist, or when the module can have multiple clients.
monitor	A monitor service is in a sense private to the module, since it cannot be requested: it is launched automatically when the module is launched, after the startup logic ends (see section <>). Once terminated, it does not return any data. Event/notification handlers cannot be set as monitors since they are requested in a different fashion.	-For initiation and configuration of some component (initiating certain hardware or software driver, for example).
permanent [[**absolute**] /time_value/]	A permanent service does not ever terminate, thus it cannot have input or output data. Once it is requested (or launched in the case of a monitor service) and its logic terminates, it is initiated again, and this behaviour is automatically repeated after each iteration. Only when the module is shut down, and before the preending logic starts (see section <>), the service is signalled to terminate when the current iteration ends. Event/notification handlers cannot be set as permanent, since that would contradict their way of being requested. If this characteristic is parameterized by a time value (greater than zero), the service will initiate the next iteration after the time indicated passes. If the **absolute** keyword is used, the time indicated will include the time already spent by the execution of the last iteration of the service.	-Usually, this characteristic is used together with **monitor** for setting periodic tasks (supervisory code, simulation, periodic event signaling, etc.). -If the task is intended to start on demand and not with the launching of the module, the permanent characteristic should be applied to a non-monitor service. -If the time value parameter is used with the **absolute** keyword, the service will be enabled as a hard real-time task with that period.

example, consider a service for displaying remote information in a SCADA system. Dynamic priority services run at the same priority that their caller. Finally, numeric prioritized services (only monitors and event/notification handlers can be set that way) are assigned a relative priority inside the priority range associated to the module, which in turn will depend on the priority bases associated to other modules during deployment (see section <>).

All services in a module may have input and/or output data that must be appropriately typed. Also, the **Data definitions** section may include data types that may define the module for other modules and for itself. Thus, other modules in the same application can use directly these data definitions as their own, just qualifying them with '::' syntax. In order to keep the structural design of a module completely portable and separated from heterogeneous concerns, H uses a common and well-defined definition language for specifying data types. It is called HDL (for "Heterogeneous Data Language"), and it is simply a subset of the OMG's IDL [<ref>] that includes only definition of types and constants (it does not consider exceptions, modules, interfaces, inheritance, etc.). This especification language is the same for all the modules, independtly on

their implementation. However, when a codification is written for a given module and a particular codification language is chosen for its implementation, the HDL can be restricted. Consider for example a codification of a module that is intended to be implemented in assembler or executed in a PLCs. It is common in that case that no real numbers or complex data types exist. If during the generation of the implementation it is found that the codification language restricts the HDL of its module, a check is performed for assuring that all the types used in the codification satisfy the restrictions imposed by the codification language, producing a generation error if not.

Finally notice that the input or output data of services may include explicit information about the semantics of the data (namely, range and units, as

shown in fig. <>). We have found this feature most useful for facilitating the reuse of modules by different programmers at different moments.

Codification Design. Apart from the structural design of a module, some codification (programming) of it must be provided to construct an executable program. Fig. <> shows the syntax for codifications, which follows an Active Object framework. A codification is always associated to some structural design, that is, to some module, either by linking it explicitly to a module name via the **`implements`** keyword or by referring to another codification via simple inheritance (the **`reviews`** keyword). In case of inheritance, the Startup logics of all the codifications in the inheritance hierarchy are assumed to execute downwards (first the most abstract), while the Preending and Postending logics are assumed to be executed upwards (first the most concrete). Service logics can be set to execute in the most convenient direction, or to substitute completely the codification of the parents if the **`replace`** keyword is used.

Figure <>

Module codification design <codification name> [**implements** <module name> | **reviews** <codification name>]

Description: **"**Description of the codification**"**

Author: "Author name"

[**Codification language:** /codif_language/]

[**Internal status**
Data definitions (types, variables, or constants; no code allowed) for the internal status of the module, written in the codification language.
End internal status]

[**Replication**
This logic is called in passive replication only whenever the leader replica has transmitted to this one its internal status. It is conceived for updating other pieces of data apart from the internal status, or to respond to errors in that replication.
[**Deportabilization:** /list_of_ids/]
End replication]

[**Startup logic** [, **Timing** /timing_range/]
Code that will be executed at the start of the module and before any service is available; written in the codification language.
[**Deportabilization:** /list_of_ids/]
End startup logic]

[**Preending logic** [, **Timing** /timing_range/]
Code that will be executed when the module is shutting down but before all the running or pending services have ended.
[**Deportabilization:** /list_of_ids/]
End preending logic]

[**Postending logic** [, **Timing** /timing_range/]
Code that will be executed when the module is shutting down and all the running or pending services have ended.
[**Deportabilization:** /list_of_ids/]
End postending logic]

[**Auxiliary logic**
Diverse pieces of code (and only code, no data) that can be useful for several of the other logics. .
[**Deportabilization:** /list_of_ids/]
End auxiliary logic]

[**Externals**
[[**Linkable:** /list_of_paths/ ;] |
[**Processable:** /list_of_paths/ ;] |
[**Passive:** /list_of_paths/ ;]]+
End externals]

(**Service** <service name> [[**replace** | **upwards** | **downwards**]] [, **timing** /timing_range/]

Here comes the internal codification of the service, written in the codification language. Possibly executed concurrently with other services' logics (depending on the service chararcteristics).

[**Replication:**
This logic only can be used in active replication. It will be called only on one replica (the last to finish the request) after all the existing replicas of the module finish a request for this service, and will have as parameters the same input and output parameters as the service.
This logic is in charge of retrieving the results produced by all the replica requests and producing a unique result through some merging policy or algorithm.

```
            The output parameters corresponding to each replica will be accessable through an array-like construction of the codification
            language, indexed by the order of the replica.
            If this logic is not included and active replication is selected, the results returned by the service are the ones of the last replica
            (the others are discarded).
          ]

        [Deportabilization: /list_of_ids/]

    End service <service name> )+

  { Event handler <signal id>

            Here comes the internal codification of the event handler, written in the codification language. It will be executed concurrently
            with other services' logics.

            In case of active replication, all the handlers will be called in all the replicas, but no special code is needed to retrieve a common
            result since handlers do not return anything.

        [Deportabilization: /list_of_ids/]

    End event handler <signal id> }

  { Notification handler <signal name>

            Here comes the internal codification of the notification handler, written in the codification language. It will be executed
            concurrently with other services' logics.

            In case of active replication, all the handlers will be called in all the replicas, but no special code is needed to retrieve a common
            result since handlers do not return anything.

    End notification handler <signal name> }

End module codification design <codification name>
```

See in the figure how the codification design includes portions of code and data written in some codification language. This is the main reason why we call H a "metalanguage": it allows the designer to cope efficiently with the existing heterogeneity in programming languages. All the so-called "logics" are sequential routines, and their variables, constants, or type definitions are considered under their scope only. They have also access without restrictions to the internal status of the module. In addition, the **Externals** section permits the designer to write portions of code in external files for make the codification clearer. These files can fit in three categories: **Linkable** for those that must be included in some way in the source code of the program generated from the codification (for example, header files in the case of using C as the codification language); **Processable** for those files that must be processed in some way after including them in the generated program (for example, for compiling or linking to the executable); and **Passive** for those that simply are to be copied in the destination directory of the generated program (for example, for configuration files). The designer should assure that the external files are completely portable in order not to compromise the heterogeneity management of H.

The codification of service logics must be able to deal with input and output parameters whose types are specified in the structural design of the module in HDL. Therefore, when a codification design is used for producing the source code of a executable program written in the specified codification language, some code must be included by the generation tool (see section <>) to transform from HDL into data types and operations in that language, and vice versa. This mapping is carried out differently for each codification language. It only depends on the codification language, even when a communication platform such as OMG CORBA [<ref>] includes IDL facilities. The general rule is that the input or output parameters of services maintain their names (as variables, typically) when mapped to the particular language. If within a given logic some variable of a type declared in the **Data Definition** section of a module is needed, the way to refer to that type can also be found in the mapping of HDL for the given codification language. Finally, also remember that a given codification language may impose restrictions on the HDL used for the structural design of the module, as explained in section <>.

For keeping the codification design as portable as possible, many constructs that would made the logics to depend on some particular component (for example, operating system facilities, user interaction, etc.) or on intrinsic features of H are dealt with through the so-called "non-portable atoms". A non-portable atom is a macro with some parameters that can be included at any point in a logic using a special syntax:

```
#_H-atom(<atom name>,…<parameters - either
constants or variables of the codification
language>…)_#
```

That syntax is intended to keep it distinguishable from the sentences written in any codification language. H provides the programmer with atoms for

dealing with many basic issues related to intrinsic aspects of H and with our general platforms. A few relevant ones are shown in fig. <>. Notice that the set of atoms that can be used in a codification may be restricted by the codification language that is chosen. For example, those for dealing with multithreading synchronization (semaphores) are not available in cyclic executive platforms.

Table <>

Atom name	*General Platform*	*Use*
Request-synchronous-static	**communication**	Issues a request for some service of a module (this includes requesting services of the very module where it is used), passing input parameters and retrieving output data if present. Errors are also communicated to the caller in special return variables. The synchronous behaviour implies that the caller is blocked until the callee returns an answer (or an error). The static behaviour implies that both the callee module and service are known before execution time. If there are several communication platforms (see section <>) available for issuing the request, they are used sequentially until one of them successfully sends and receives the information (the atom does not allow the programmer to specify one platform to use). Also, real-time requirements can be included in the atom for specifying the minimum and maximum times allowed in the request, and also to set a timeout. If the timeout fires, the request is discarded (as any future incoming answer). In the case of the callee module being a repetition (see section <>, paragraph on implementation), this atom also can include the repetition identifier to refer to a particular copy of the destination module. In the case of the callee having several replicas for fault-tolerance purposes (see section <>, implementation), the request will be issued to all of them; in the case of active replication, they will coordinate to send a single result.
Request-synchronous-dynamic	**communication**	The same but with a dynamic behaviour: both the callee module and service can be set up at execution time.
Send-event **Send-notification**	**communication**	Both issue asynchronously a signal to some module, which can be set dynamically at execution time or as a constant in design time. Due to their asynchronous nature, these atoms do not allow the programmer to specify real-time requirements. Events have input parameters, so this can be used as a general asynchronous communication method. No response is received by the caller or guarantee that the callee has caught the event. Notifications are intended only for internal events (in the same module as the caller), so they do not carry the destination module information.
Number-repetitions **Name-repetition**	(intrinsic)	If the codification is a repetition of a given module (see section <>), these atoms allow it to check out how many repetitions exist of its module and also get there names, in order to construct dynamic requests which are less tied to a particular application than static ones.
Real-time-suspend **Time-stamp**	**real-time**	Perform an operating-system dependent suspension of the thread that executes that logic during a given time, or read the current time, respectively. The latter provides a global time shared and synchronized through all the computing machines of the application if a real-time platform with that capability is used. Otherwise, it provides a local measure of time.
User-log	**real-time**	Records a time-stamped user log that can be examined off-line if the codification has activated its logging option (see section <>).
Critical-zone-create **Critical-zone-enter** **Critical-zone-leave**	**execution / hardware**	These atoms allow the programmer to synchronize multiple concurrent threads (that is, reentrant services as well as monitor services and event/notification handlers) for accessing shared resources, mainly the internal status of the module. They follow the classic semaphore paradigm. They may include timeout parameters for real-time needs.
Drop-replica	**fault-tolerance**	Drops the current codification as a replica, both in passive and active replication scenarios, even when it is the only replica currently up (in that case the module will cease to respond). It is intended for reacting to situations where a given replica is not able to continue working correctly.

It is remarkable that all the logics of a codification include an optional section called **Deportabilization**. If a logic needs some software or hardware component not covered by the non-portable atoms (for example, a software library for mathematical processing or a data acquisition card), that section encourages the designer to make it explicit, and so, provides a clean and effective way of including software and hardware heterogeneity at the codification level.

Codification design also permits to specify real-time requirements (through the **Timing** keyword), mainly for validation purposes.

The Non-Portable Components

Both the structural design and the codification design of modules explained in the previous section are mostly portable. In a heterogeneous application, however, it is common to have a number of dependencies that cannot be avoided, and in fact, must be exploited. We have already seen that in the codification design weak dependencies can appear explicitly through the **Deportabilization** section. In order to use correctly that feature, the non-portable components must be previously defined. In our framework this is enabled through General and Particular Platforms. General Platforms serve to classify the non-portable support necessities of the application: hardware, execution, communications, etc., and allow us to classify both commercial and non-commercial off-the-shelf solutions. Particular platforms are instances of general platforms (for example, a given operating system or a concrete processing hardware). Currently, H recognizes the following general platforms:

a) *Hardware platforms* (HP). A HP represents a set of hardware devices needed for the physical execution of the application, which includes at least one processor unit and shares a motherboard[1]. This kind of platform can group together: CPU(s), motherboard devices (hard disk, sound card, graphic card, real-time clocks, etc.), plugged-in devices (acquisition boards, automation interfaces, network interfaces, etc.), and peripherals (monitor, printer, external storage devices, etc.).
b) *Execution platforms* (EP). An EP represents the basic software execution environment of the application. This includes: operating systems, virtual machines, software libraries and execution libraries (e.g., interfaces with hardware devices).
c) *Communication platforms* (CP). A CP comprehends the software needed for communicating the modules of a distributed application. Its particular platforms can provide from simple protocol support (peer-to-peer, TCP/IP) to more abstract object distribution (like CORBA [22]), even a monolithic scheme (that is, no network).
d) *Real-time platforms* (RTP). A RTP provides real-time facilities in software form: real-time scheduling, time measurement, synchronization, etc. It can also provide clock synchronization mechanisms for giving to the designer of the application a common time measurement in all the computing machines.
e) *Fault-tolerance platforms* (FTP). A FTP provides software fault-tolerance facilities (for example, active replication [<ref>]). In H there is a general support for including the facilities provided by these platforms.

Fig. <> shows the syntax for defining particular platforms. If a codification design refers to some particular platform in one of its **Deportabilization** sections, and that platform is not defined in any such file, no application that uses that module can be constructed.

Figure <>

```
Particular /general_platform/ platform
<platform name>

   Description:  "Description of the
platform"

End particular platform <platform name>
```

Communication platforms have some special issues that we should remark. Firstly, an application can use more than one communication platform for issuing service requests and signals between modules. A module's codification that is set to run on a given execution/hardware platform will have access to the communication platforms declared to be supported by that execution/hardware (see section <> for a deeper explanation on how to specify these support relations between platforms). This means that the codification is automatically provided with as many message-reception loops as communication platforms has available, all running concurrently and serving requests and signals to the module. Also, when a codification issues a service request or a signal, it must choose a communication platform if more than one is present. In that case, H only guarantees to issue the request through the first platform that is able to finish it correctly. If some platform fails, another one (without any predefined ordering) is selected and the request repeated. If no platform succeeds, an error is returned.

A second issue with communication platforms comes from the fact that a communication link can be mixed (for example, by using a gateway that conveys data from a network of one type to a different one, like in the example of section <>). Since H is only

[1] In a microcontroller, the "motherboard" typically includes just the CPU, main memory, and I/O facilities.

intended to use communication layers above the transportation one, and the transportation layer can hide the link details effectively, there is no problem in mixing links in a network for the design of the application. The designer just has to include the communication platforms corresponding to both ends of the link in the computational machines where the modules' codifications are going to be deployed.

A third issue with these platforms is that many of them require that the modules are registered in some directory service (the Name Service in CORBA, for example) in order to be visible to others. In H this is guaranteed to be done dynamically, that is, each time a service or event request is initiated, the program will check if a reference to the destination module has been previously obtained through that directory service; if not, the reference will be get at that time, otherwise the previous reference will be used. Therefore, possible dead-locks in retrieving these references (for example a cyclic situation where a module is going to request services from another one which in turn will request services from it) are avoided. Appropriate mechanisms for deleting references of modules that are not longer running must be provided by the communication platform.

The Application

Once a repository of modules is available with both structural and codification designs, whole applications can be designed. From the structural point of view, an application is just a set of modules that communicate to each other through service requests and/or events. From the codification point of view, it is a set of relations between programs (generated from codifications) and computational machines (where those programs run). Since an application can be specified at both levels, two parts must be included in its design, as explained in the next subsections.

Application. For constructing the structure of an application, a list of modules (its structural design) must be provided. This is done through the syntax shown in fig. <>.

In this specification there is a special feature to cope with the need of having more than one copy of the same module in the application, for example, when the application includes a number of identical programmable devices. For activating that feature, the **`repeated`** keyword can be used after a module name, providing at least two identifiers. Each identifier will refer to a copy of the module and must be different from the others. Notice that this feature is independent on and has nothing to do with replication, which is used for fault-tolerance purposes as explained further on.

Figure <>

```
Application <application name>

   Description:  "Description of the
application"

   Author: "Author name"

   Modules: /list_of_repmodules/

End application <application name>
```

Implementation. Each structural design of an application must be accompanied by a specification of its implementation in order to produce executable programs. In fact, several implementations of the same application can be provided, for example if there are changes in the distribution of the modules among the computers, or in the hardware support, etc.

An implementation of an application is written following the syntax shown in fig. <> (details on the syntax of some constructs can be found in Appendix <>). The first thing to notice is that a number of specific instances of particular platforms must be provided. For example, consider an implementation of a control application with a number of PLCs: several instances of the controller platform must be specified.

Figure <>

```
Implementation <implementation name> for
<application name>

   Description: "Description of the
implementation"

   Author: "Author name"

   Platforms: /list_of_platform_instances/

   Support: /list_of_supports/

   [ Fault-tolerance: /list_of_FTs/ ]

   Deployment: /list_of_deployments/

End implementation <implementation name>
```

In the implementation design also appears a **`Support`** section. This is the way of specifying the support relations existing between particular platforms (more exactly, between instances of particular platforms). For example, a given computer can provide support for a given operating system, which in turn can provide support for a given processing library. These relations form an acyclic graph which allows us to validate if the application to be implemented can rely on the appropriate components to satisfy its requirements.

The optional **`Fault-tolerance`** section allows the designer to activate fault-tolerance mechanisms in the application through replication, either active or passive for each module. In particular, it associates an instance of a fault-tolerance particular platform (which must have appeared previously in the **`Support`** section or be either the **`built-in`**

platform) to any module for which replication should be activated. All the replicas of that module will use that type of replication (active or passive) and that platform to coordinate their service requests and internal status.

The **`Deployment`** section allows the designer to distribute codifications of the modules among the available execution/hardware platforms declared in the **`Support`** section. There are two features that must be contemplated: on the one hand the possibility of deploying a number of repetitions of a given module which are distinguishable as different modules by the identifiers previously declared (with the **`repeated`** keyword) in the application file; on the other hand the capacity of deploying different replicas of a module for fault-tolerance purposes. H has provision for the following cases:

a) A module is deployed as a single executable program (no repetitions of the module and no replicas). In that case, the **`repetition … of`** keyword must be omitted and there should appear only one codification corresponding to the module in the list of deployments.

b) A module is deployed with replication (fault-tolerance activated), but without repetitions (it appears as only one module to the rest of the application). In that case, the **`repetition … of`** keyword must also be omitted, but a number of *identical* codifications of the given module should be declared in the list of deployments, and all of them will be either active or passive replications (according to the **`Fault-tolerance`** section explained before). In passive replication, the **`replica`** number will define the order in which the replicas will be selected for processing requests (when the highest number ceases to respond after the timeout given in the **`Fault-tolerance`** section, the next one will take its place; whenever the selected replica produces the result, its internal status will be replicated in all of the others; all the replicas not receiving correctly that communication will be deactivated automatically). On the contrary, in active replication, the **`replica`** number will be used to index the output data produced by each replica in the code in charge of merging/coordinating their results. In that case the replica to produce the final result will be the latest in finishing the request (no communication is needed to replicate the status since all the replicas process the same data; however a broadcast mechanism is needed to send the request to all of them)[2].

c) A module is to be deployed with repetitions that appear as different modules for the rest of the application, some of them also including replication for fault-tolerance purposes. Then, the **`repetition … of`** keyword must be added to all the codifications of the module. Those repetitions that include replication will appear as several codifications that share the same repetition identifier. The codifications that only appear with one repetition identifier will not have fault-tolerance support.

Producing, Executing, Debugging, and Maintaining Applications

Once the design and implementation of the application is written, automatic tools that validate the design and, if possible, construct the executable programs are available.

For automatically generating the programs of the application (that can be many if they are intended to run on different machines over a network), we have developed a software called *H-apc*, for "Heterogeneous APplication Constructor", that interprets the files described in the previous subsections and generates source code for the specified codification languages, execution platforms (operating systems), and a number of compilers and interpreters. In particular, it generates a set of source code files and locates them in a separate directory for each codification. *H-apc* is strongly based on code templates in order to be easily extended with new languages, OSs, and compilers/interpreters. Currently we have a version of this tool integrated into a visual CASE application for designing modules and codifications, called the BABEL Module Designer [<ref>]. It provides support for codifications written in C, C++, JAVA, and for these execution platforms: MS Windows NT+, LynxOS, and the JAVA VM.

For deploying and executing a given application, once it has been constructed with *H-apc* and compiled/linked if needed, we use another software, called *H-apx* (for "Heterogeneous APlication eXecutor"), that is able to launch the programs on their respective computational machines from a remote station, if it can, or locally, and collect their logging information when that option is activated through the **`logging`** keyword in the list of deployments of the implementation file. The list of deployments also configures the relative priorities of the codifications (the **`priority`** keyword) and their launching order (the **`order`** keyword). Pauses

[2] Notice that active replication is suitable for modules that do not request services from other modules (or from theirselves), since in other case all the replicas will issue the request, generating a not proper operation. For modules that request services, it is more suitable to use passive replication instead.

before the execution of each deployed codification can be set (the **`pause`** keyword). When a codification produces an executable program, it may need some command-line arguments to be sent. This can also be achieved through the **`cl-arguments`** option. All these parameters are currently entered in a visual application called the BABEL Execution Manager [<ref>], that can be deployed on MS Windows computers.

If logging has been activated for some codification, the results can be analyzed off-line through our *H-apl* ("Heterogeneous Application Logger"), a software that shows graphically the sequence of events that have occurred during execution (a previous version was the BABEL Debugger [<ref>]). This allows the programmers to inspect possible real-time flaws or bad ordering of requests, apart from user-defined events that can be registered at any time. The logging time-stamps are global (synchronized between all the computing machines of the application) if real-time platforms are available that provide such feature; otherwise logs are time-stamped locally for each codification.

Finally, all the designs produced with H can be stored in our BABEL web site ([<ref>]) for maintainance. The site includes multi-user priviledged accesses, version support, and some validation tools. For example, it is possible to examine a set of modules of an application to detect loops in their service requests or the impossibility of satisfying the real-time requirements specified in the **`Timing`** keywords of codifications.

A Case of Study

Figure <> is a case of study of a simple industrial cell in charge of classifying goods that are transported on a conveyor. The desired operation of the system is as follows: a good on the belt is transported until a presence sensor detects it, which triggers the detention of the belt and the recognition of the object. The recognition system (composed of two intelligent cameras for improving recognition) captures an image and classes the object according to its brightness. The result of the classification makes the routing mechanism (a diverter on the belt) to head the object to the correct direction. In the following subsections we describe in more detail the hardware and software available in this plant and the design and implementation of a control and SCADA application using H and the H tools. We will show how, in spite of the apparent simplicity of this plant, the heterogeneity level can be quite high.

Hardware and Software Available

At the field level, the plant has the following sensors, actuators, and controllers:

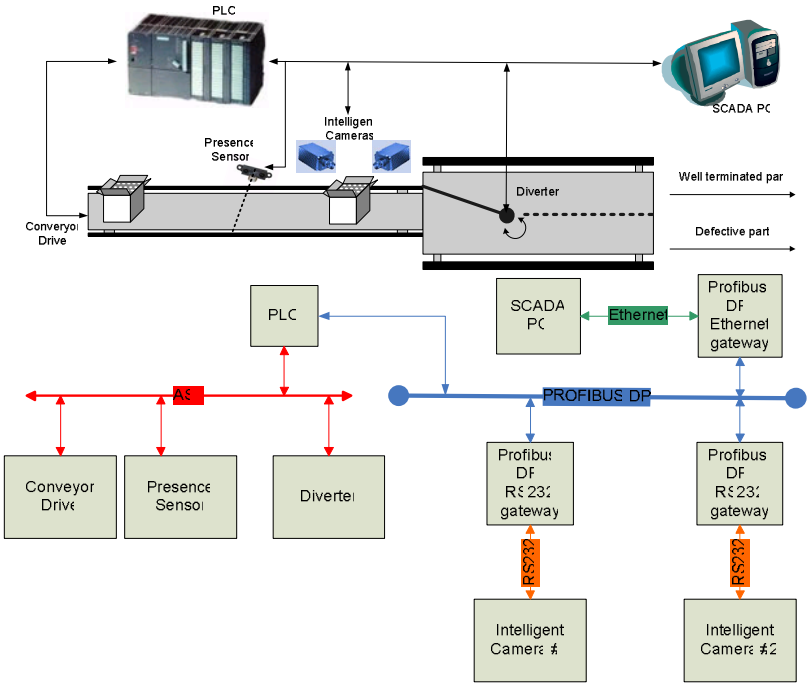
a) A presence sensor (typically a photoelectric sensor, such as [<ref>]) that gives an on/off binary output under the presence or not of a good on the belt.

b) A conveyor drive, composed of a gearmotor and a starter (for example [<rf>]) attached to the pulley of the conveyor belt. It has no reverse operation and no speed regulation (just stop/start through the starter).

c) A pneumatic diverter (such as [<ref>]) that admits an on/off signal for selecting one out of two lanes in the second segment of the belt.

d) Two intelligent B/W cameras with embedded processors with capturing and basic image processing capabilities (for example, [<ref>]). Each one can be triggered to start the capture and provide two bits output indicating the presence of a well terminated part (00), a defective part (11), or a not-known or

Figure <>



error status (01/10). The use of two identical cameras is for providing a consensus between both results and so to diminish the probability of a recognition failure. Only one of them will provide the final result to the controller.

e) A PLC controller. For this case study we will consider an out-of-catalog device in order to augment heterogeneity and also illustrate the typical reuse of old components in many manufacturing environments. In our example it is the modular Simatic S5-100U controller [<ref>].

At this level, the devices are connected through a heterogeneous network:

- Non-programmable sensors and actuators (presence, drive, diverter) are connected through an AS-i bus to the PLC (in the case of the Simatic S5, this must be done through an AS-i module [<ref>]).

- The controller is connected to the cameras as a master in a Profibus-DP bus [<ref>] (a CP 5431 module for this PLC is needed [<ref>]). This hard real-time bus will also be used for connecting the field to the SCADA system. Profibus classifies the devices connected to it as masters or slaves (there can be several masters, with a maximum of 32 devices if no repeater is used). Masters can initiate communication, but slaves can also transfer data in a peer-to-peer fashion.

- The VS710 cameras chosen for this case of study provide direct connection to Profibus-DP, but for increasing heterogeneity, we will consider their RS-232 connections instead. So, we need a pair of Profibus-DP / RS-232 gateways (for instance, [<ref>]) that set the cameras as slaves in Profibus.

The enterprise level will contain a number of conventional PCs. One of them will include a SCADA software. We will consider that all these PCs share an Ethernet network, and the SCADA computer will pass through to the field by means of a Ethernet / Profibus-DP gateway [<ref>], being seen as a master from the Profibus perspective.

Concerning the software, there are three programmable devices in our example:

- Both intelligent cameras include an Intel 486 processor @ 100MHz [<ref>] with 16Mb of main memory. They run MS-DOS 6.22 [<ref>]. The camera buffer can be dumped to main memory through suitable calls to a software library. The development is carried out on a different computer (using Visual C++ 6.0 [<ref>] if C or C++ languages are chosen) and the binary files transferred to the embedded processors via RS-232 or the other available connections. Once the program is in the camera, it operates autonomously.

- The PLC has a CPU with a well-defined execution time for its instructions. It also includes a ROM with a cyclic executive that starts in each cycle by transferring an image of all the inputs and outputs of the PLC to main

memory, follows by executing the program, and ends by transferring back the input/output data to the devices. The program can be written in three modes. We are interested in AWL [<ref>], which is a textual programming language similar to assembly, with no real number processing capabilities. The program is developed in an external computer and transferred to the PLC via a serial cable. Once there, it can be stored in an EPROM. With respect to the communication modules, the AS-i module appears to the CPU as a set of binary data in the input/output space, while the Profibus-DP provides a number of predefined functions that can be called for performing basic SEND/RECEIVE operations.

- The PC in charge of running the SCADA front-end will run a MS-Windows XP OS [<ref>], which can be programmed in a variety of languages. This computer will serve both as a development computer (not only for their own programs but also for compiling and transferring programs to the PLC and the cameras), and as an execution environment. We will choose JAVA [<ref>] for programming our front-end due to its graphical capabilities and portability among platforms. The Ethernet connection can be accessed through socket interface [<ref>] and a Profibus-DP library provided by the gateway manufacturer.

Designing the Application with H

In the design of this application with the H language we can distinguish three different modules, intended for managing each of the programmable devices: a *PLC Control* module, a *SCADA Front-End* module and an *Inspection* module. The PLC Control module will be in charge of properly managing the conveyor actuators based on the readings of the proximity sensor and the recognition performed by the cameras. The Inspection module will reside in the cameras and provide object recognition facilities. The SCADA Front-End module will continuously retrieve information from the whole system to be graphically shown in the operator console, and detect possible failure situations. Figs. <> show the structural designs for these modules. Notice the object-oriented inheritance for future reuse of the designs. The PLC Control module has some hard-real time constraints in the form of a cyclic task in charge of sequencing the shop plant. Also notice that the SCADA Front-End receives information from the plant asynchronously, without hard time requirements (it also is able to act on the plant to stop or start its operation under user demand). No information depending on a given hardware or software is needed, and thus the portability and reusability of these structural designs is maximum.

Figure <>

```
Module abstract structural design FieldDevice

    Description: "An abstract module that
encapsulates a general field device"

    Author: "The H Team"

    Data definitions
        enum DeviceStatus
{ok,stopped,failure};
    End data definitions

    Service GetStatus
        Characteristics: reentrant;
        Priority: dynamic
        Outputs: DeviceStatus st;
        Description: "Return the current
status of the device"
    End service GetStatus

End module structural design FieldDevice
```

Figure <>

```
Module abstract structural design
ControlDevice inherits from FieldDevice

    Description: "An abstract module that
encapsulates a general control field device"

    Author: "The H Team"

    Data definitions
        enum DeviceStatus
{ok,stopped,failure};
    End data definitions

    Service NumberOfDevices
        Characteristics: reentrant;
        Priority: dynamic
        Outputs: long num;
        Description: "Return the number of
devices controlled by this"
    End Service NumberOfDevices

    Service GetStatusOfDevice
        Characteristics: reentrant;
        Priority: dynamic
        Inputs: long numdev; // Index of
device to check, from 0
        Outputs: bool invaliddev,
FieldDevice::DeviceStatus st;
        Description: "Return the current
status of the indicated device"
    End service GetStatusOfDevice

    Service GetStatusOfDevices
        Characteristics: reentrant;
        Priority: dynamic
        Outputs:
sequence<FieldDevice::DeviceStatus>;
        Description: "Return the current
status of all the devices controlled by
                    this, including itself
at the end of the sequence"
    End service GetStatusOfDevices

    Service SuspendControl
        Priority: prioritized;
        Description: "Suspend the execution of
the system controlled by this"
    End service SuspendControl

    Service ResumeControl
        Priority: prioritized;
        Description: "Resume execution of the
system if it is in suspension"
    End service ResumeControl

End module structural design FieldDevice
```

Figure <>

```
Module structural design PLCControl inherits
from ControlDevice

    Description: "Control of the sensors and
actuators of a case of study shop floor
application"

    Author: "The H Team"

    Service MainControl
        Characteristics: monitor, permanent
absolute 500 milliseconds;
        Priority: prioritized
        Description: "This service is a cyclic
hard-real time task in charge of checking
                      the status of the
presence sensor and stop/start the conveyor
drive
                      and the diverter as
necessary after consulting the Inspection
module"
    End service MainControl

End module structural design FieldControl
```

Figure <>

```
Module structural design Inspection inherits
from FieldDevice

    Description: "Inspection of parts to
decide if they are right or defective"

    Author: "The H Team"

    Data definitions
        enum InspectionResults
{ok,defective,error}
    End data definitions

    Service Inspect
        Priority: dynamic
        Outputs: InspectionResults result;
        Description: "This service performs an
inspection of the part"
    End service Inspect

End module structural design Inspection
```

Figure <>

```
Module structural design SCADAFrontEnd

    Description: "Graphical Front-End for the
SCADA system"

    Author: "The H Team"

    Signal definitions

        signal PartDetected
            description: "Signalled when a new
part has arrived on the belt and
                          the recognition
system has decided of which type it is"
            parameter:
Inspection::InspectionResults newpartinsp;
        end signal PartDetected

    End signal definitions

    Service GraphicalInterface
        Characteristics: monitor, permanent;
        Priority: unprioritized;
        Description: "This service is in
charge of creating and maintaining the"
                      graphical interface for
user interaction"
    End service GraphicalInterface

    Service RefreshDevicesStatus
        Priority: dynamic;
        Description: "Consult the current
status of the system and put it on the
                      graphical front-end"
    End service RefreshDevicesStatus

    Service RefreshStatistics
        Priority: dynamic;
        Description: "Put on the graphical
interface up-to-date statistics on the
                      parts that the system has
processed"
    End service RefreshStatistics

    Service SuspendSystem
        Priority: prioritized;
        Description: "Stop the whole system
temporarily"
    End service SuspendSystem

    Service ResumeSystem
        Priority: prioritized;
        Description: "Resume the whole system
activity"
    End service ResumeSystem

    Event handler PartDetected
        Priority: 10
        Description: "Adds one to the correct
parts or to the defective parts, or increments
                      the number of failures (in
that case it also stops the system), depending
                      on the type of the part
detected"
    End event handler PartDetected

End module structural design Inspection
```

The next step is to design the codifications for these structures. This is when heterogeneity must be dealt with for the first time. On the one hand, specific programming languages must be selected. We will choose JAVA for the SCADA Front-End, C for the embedded camera processors, and AWL for the Simatic PLC. On the other hand, some information on the particular platforms that are needed for executing the codifications may be included. Notice that through the use of H-atoms, most of the portability/heterogeneity parts of the codification will still remain portable. Figs. <> show some fragments of the codifications for the three modules. The Inspection module includes active replication for merging the results of both cameras. The PLC MainControl service is deportabilized by the CP 2433 module (the one that provides access to the AS-i bus from the PLC), since it is used to access field devices that do not contain modules. However, the CP 5431 needed in the PLC for communicating through the Profibus does not deportabilize the codification: it is used as a communication platform between codifications, and thus, it could be substituted in the future without compromising the logics of fig. <>.

Figure <>

```
Module codification design PLCControlS5 implements PLCControl

    Description: "Codification of the PLC Control module for executing in a Simatic S5 PLC"

    Author: "The H Team"

    Codification language: step-5-awl

    Internal status
        {-{
                DB2

                   KY = 1   #state of the system 1 -> ok, 0->failure, 2 -> stopped#

                BE
        }-}
    End internal status

    Startup logic // This logic will be dumped to the OB 21 block of the S5 by the H tools
        {-{
                    S M 1.0
                    S M 1.1
                    R M 2.0
                    R M 2.1
                        ...
                    = A 0.1    #start moving the conveyor#
        }-}
    End startup logic

    Service MainControl
    // This logic will be dumped to OB 1, which is executed cyclically by the S5
        {-{
                    C DB 2
                    ...
                    #calling the inspection service. Results are stored in two marks#
                    #_H-atom(Inspection,Inspection,"M 1.0,M 1.1")_#
                    U M 1.0
                    U M 1.1
                    = A 0.2   #it moves the diversion device to 1 if the result is 11, or 0
otherwise#
                    O(
                    U M 1.0
                    UN M 1.1
                    )
                    O(
                    UN M 1.1
                    U M 1.0
                    )           #result is 01 or 10, that is, a failure in the system#
                    T BI 1
                    = A 0.1     #the conveyor is activated if system is ok#
                    ...
                #The rest of the code#
            }-}
        Deportabilization: SimaticPLC // Since the codification cannot access the non-programmable
devices without it
    End service MainControl

    // The other services written here …

End module codification design PLCControlS5
```

Figure <>

```
Module codification design VS710CameraInspector implements Inspection

    Description: "Codification of the inspection logic for the VS710 camera"

    Author: "The H Team"

    Codification language: ansi-c

    Service Inspect, timing 10 milliseconds .. 100 milliseconds
      {-{
        // Accesses to the camera
        ...
        // Dump image into main memory
        ...
        // Run recognition process
        ...
      }-}
      Replication:
         {-{
            int res1=result[0]; // result of the service as produced by the first replica
            int res2=result[1]; // the result produced by the second replica
```

```
            if (result[0]==result[1])
                result=result[0]; // both replicas have returned the same
            else
            {
                // If one of the replicas does not work, take the other
                if (result[0]==Inspection::error)
                    result=result[1];
                else if (result[1]==Inspection::error)
                    result=result[0];
                else // both replicas disagree; take the worst case
                    result=Inspection::defective;
            }
         }-}
    End service MainControl

    // The other services written here

End module codification design VS710CameraInspector
```

Figure <>

```
Module codification design JAVASCADAFrontEnd implements SCADAFrontEnd

    Description: "Codification of the graphical front-end in JAVA with swing"

    Author: "The H Team"

    Internal status:
            {-{
                 int num_malfunctions;
                 int num_detected_parts;
            }-}
    end internal status

    Startup logic
            {-{
                        num_malfunctions=0;num_classA=0;
                        num_classB=0;
                        #_H-atom(Critical-zone-create,"1")_#
            }-}
    end startup logic

    Codification language: java-2.0

    // The other services written here

    Event handler PartDetected
        // a new part detected and potentially a stop
     {-{
        #_H-atom(Request-synchronous-static,PLCControl,GetStatusOfDevice,"motor-
drive,error,result")_#
        if (!error)
        {
            if (result!=ok)
            {   #_H-atom(Critical-zone-enter,"1")_#
                num_malfunctions++;
                #_H-atom(Critical-zone-leave,"1")_#
            }
            else
            { #_H-atom(Critical-zone-enter,"1")_#
              num_detected_parts++;
              #_H-atom(Critical-zone-leave,"1")_#
            }
        }
        //Refresh graphical front-end…
      }-}
    End event handler

End module codification design JAVASCADAFrontEnd
```

Table <>

Particular Platform	*General Platform*	*Description*
SimaticPLC	Hardware Platform	The Simatic S5-100U (CPU 103) PLC basic hardware and operating system. Also the CP2433_ASi module for accessing the AS-i devices.
SimaticPLCTiming	Real-Time platform	Hardware and software support for the real-time features of the simatic PLC.
CP5431_Profibus	Communications Platform	Simatic module for communications through Profibus-DP
SimaticProfibus_Broadcast	Fault-Tolerance Platform	A software library for performing broadcast operations from the Simatic-Profibus side, needed in the active replication of the intelligent cameras.
VS710Camera	Hardware Platform	The hardware of the VS710 intelligent camera, including the processor, memory, and internal connection to the CCD sensor.
MSDOS622	Execution	The operating system for the VS710 camera embedded processor, including the

	Platform	software libraries for accesing the camera.
RS232_MSDOS622	Communication Platform	A software communication platform for issue service and event requests through an RS232 connection on a MS-DOS 6.22 OS.
PC	Hardware Platform	The conventional PC in charge of the SCADA front-end.
JAVA_VM	Execution Platform	The JAVA Virtual Machine for executing the Front-End.
EthernetProfibusGateway	Comunication Platform	Hardware and software platforms for providing support for Profibus communications through Ethernet gateway.

In addition to the design of modules and codifications, the particular platforms involved in the application must be enumerated for an implementation to be produced. Since the ".ppl" files are really simple, we rather give a table with the platforms of our example (table <>).

The application can now be designed, as shown in fig. <>. A possible implementation is shown in fig. <>.

Figure <>

```
Application CaseStudyApplication

    Description: "Design of the application
from a structural point of view"

    Author: "The H Team"

    Modules: PLCControl, Inspection,
SCADAFrontEnd

End application CaseStudyApplication
```

Figure <>

```
Implementation OurImplementation for
CaseStudyApplication

    Description: "An implementation that uses
the codifications given previously"

    Author: "The H Team"

    Platforms:
        AnOldS5 is SimaticPLC,
        S5Timing is SimaticPLCTiming,
        AProfibusModule is CP5431_Profibus,
        BuiltinBroadcast is
SimaticProfibus_Broadcast,
        Camera1, Camera2 are VS710Camera,
        Copy1OfDOS, Copy2OfDOS are MSDOS622,
        CRS232Library1, CRS232Library2 are
RS232_MSDOS622,
        EnterprisePC1 is PC,
        OurCopyOfJAVAVM is JAVA_VM,
        EPGateway is EthernetProfibusGateway

    Support:
        AnOldS5 supports S5Timing,
AProfibusModule, BuiltinBroadcast;
        Camera1 supports Copy1OfDOS,
CRS232Library1;
        Camera2 supports Copy2OfDOS,
CRS232Library2;
        EnterprisePC1 supports
OurCopyOfJAVAVM, EPGateway;

     Fault-tolerance:
        Inspection uses BuiltinBroadcast for
active replication;

    Deployment:
        PLCControlS5 deployed on AnOldS5 order
2 pause user logging on;
        JAVASCADAFrontEnd deployed on
OurCopyOfJAVAVM order 3 pause user;
        VS710CameraInspector replica 0
deployed on Camera1 order 1;
        VS710CameraInspector replica 1
deployed on Camera2 order 0;
```

Implementation, Execution, Debugging, and Maintainance with the H Tools

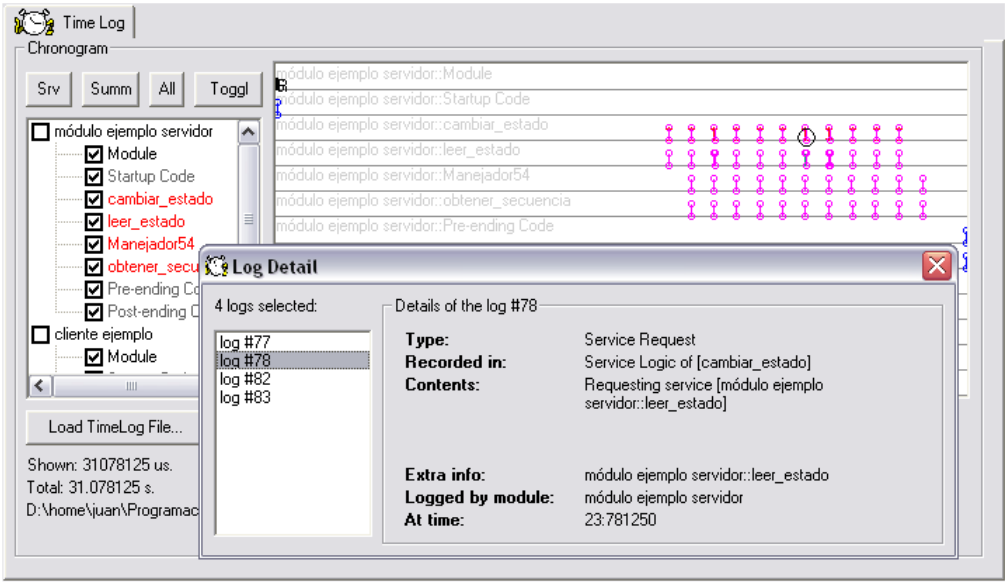
From the designs proposed previously the H tools can produce implementations. First of all, the implementation file (fig. <>) serves as an index to collect all the modules and codifications involved in the application. The H-apc tool is intended to produce the source code of complete programs from the codifications, including code for communications and support of the H-atoms (for this case of study, only the SCADA Front-End and the Inspection modules' codifications could be processed by our current implementation of the H-apc). The source code produced from each codification must be compiled/linked by some existing software related to the codification language and to the execution/hardware platform where the module's codification will be run. For example, for the Inspection modules' codifications, the source code must be compiled/linked by MS Visual C++ 6.0 for constructing a console application suitable for MS-DOS 6.22; for the SCADA Front-End, the JAVA compiler must be executed to produce JAVA bytecode programs. Once the codifications have been transformed into executable, independent programs, they must be transferred to the corresponding execution/hardware platforms. In the future we plan to develop an automatic tool to perform this deployment, but currently we do it manually.

The second H tool to use is the H-apx. It is in charge of launching the codifications by following the sequence order specified in the implementation file. Our current version would be able to do this from a centralized computer in a network using TCP/IP communications. Further extensions are needed to include remote launching in PLCs.

Once the application is terminated (which can be commanded also from the H-apx), the debugging results can be collected and passed to the H-apl. In our current implementation, H-apx is in charge of collecting all this information and producing a single log file that is then passed to the H-apl. A typical output of the H-apl is shown in fig. <> for an example of robotic application.

Finally, all the files involved in the application can be stored in our BABEL web site (currently they are compacted into a single file) for maintainance. This site also provides facilities to check some dependency problems between modules and the satisfaction of real-time requirements.

Figure <>



Conclusions and Future Work

We have presented in this paper a meta-programming framework for coping efficiently with heterogeneity in the shop floor (as long as the shop floor includes programmable devices). We have developed a number of code generators for H in the last years, and this can be extended with an unbounded number of generators in the future. Basically, the H tool that is most affected by the inclusion of new particular platforms and codification languages is the application constructor *H-apc*. Thus we have developed it based on several templates that can be easily changed and added to the tool, in most cases without recompiling it. Thus, heterogeneity in the future is guaranteed to be coped properly.

Some features that are not dealt with currently in the H metalanguage itself and may be subjected to further work in the future are: migration mechanisms for codifications [<ref>], higher levels of specification/design (for example, layers over H that are specialized in particular domains: manufacturing, robotics, etc.), more complex and mathematically grounded validation mechanisms, non-textual codification languages (for example G of Labview [<ref>], statecharts [<refs>], etc.), reflexive properties (to handle the structure of the application from within the application), automatic re-launching of failed codifications with resuming of previous internal status, integrated development environments (IDEs) with facilities for managing all the stages of the design/implementation/testing of applications and assistants for application templates, more sophisticated asynchronous communications between modules, exception handling at the design level, etc.

Appendix – H basic syntax

In this appendix we provide two figures that include the syntax of the basic elements of the H metalanguage.

Figure <>

```
list_of_ids := <identifier> | <identifier> , /list_of_ids/

HDL_data_type := <OMG's IDL data type definition>

HDL_data := /HDL_data_type/ <data identifier> [range <minimum value that can have the
                    data> … <maximum value> units <units for these values: a text
                    abbreviation, usually> ]

list_of_HDL_data := /HDL_data/ | /HDL_data/ , /list_of_HDL_data/
```

```
list_of_HDL_defs := /HDL_data_type/ ; | /HDL_data_type/ ; /list_of_HDL_defs/

time_unit := year | day | hour | minute | second | tenth | hundredth | millisecond |
                    microsecond | nanosecond

time_units := [ /time_unit/ | /time_unit/s ]

time_value := <an integer> /time_units/ | unspecified

timing_range := /time_value/ .. /time_value/

signal_def := signal <signal name>
                 description: "description and purpose of the signal"
                 [ parameter: /HDL_data_type/ ]
              end signal <signal name>

list_of_signals_defs := /signal_def/ | /signal_def/ /list_of_signals_defs/

signal_id := [<module name>::]<signal name>

serv_char := reentrant | monitor | permanent [ [absolute] /time_value/]

list_of_serv_chars := /service_char/ ; | /service_char/ , /list_of_serv_chars/

handler_prio_level := unprioritized | prioritized | <positive integer>

service_prio_level := /handler_prio_level/ | dynamic

list_of_paths := <complete path of a file> ; | <complete path of a file> ,
                 /list_of_paths/

codif_language := iso-cpp | ansi-c | java-2.0 | i8051-asm | step-5-awl …

general_platform := hardware | execution | communication | real-time | fault-tolerance

repeatable_module := <module name> [ repeated {<identifier> , /list_of_ids/} ]

list_of_repmodules := /repeatable_module/ | /repeatable_module/ , /list_of_repmodules/

platform_instance := ([,] <particular platform instance>)+ [is|are] <platform name>

list_of_platform_instances := /platform_instance/ | /platform_instance/ ,
                      /list_of_platform_instances/

platform_support := <platform instance name> supports (<platform instance name> [,])+

list_of_supports := /platform_support/ ; | /platform_support/ ; /list_of_supports/

FT := <module name> uses <particular platform instance (fault-tolerance)> for [active
                      replication | passive replication [with timeout /time_value/] ]

list_of_FTs := /FT/ ; | /FT/ , /list_of_FTs/

codif_deployment := [repetition <identifier> of] <codification name> [replica
                      <positive integer>] deployed on <particular platform instance
                      (hardware or execution)> [priority <positive integer>] [order
                      <positive integer>] [pause [ /time_value/ | user ]] [logging
                      [on|off]] [cl-arguments "text with command line arguments for
                      the executable program"]

list_of_deployments := /codif_deployment/; | /codif_deployment/ ;
                      /list_of_deployments/
```

Table <>

Syntax Symbol	*Use*
[]	Optional element (zero or one instances of the element inside the square brackets)

|	Separates exclusive options. If used inside a square bracket construct, the square brackets change their meaning to "exactly one option of the elements inside the brackets". If those square brackets are appended with a '+' (plus) symbol, then more than one of the inside elements can be selected (but at most, one of each).
…	Represents an unbounded number of options, or more precisely, a list of elements that can be extended in the future.
()+	Indicates one or more repetitions of the element inside the parentheses.
{}	Indicates zero or more repetitions of the element inside the curly braces.
//	The text inside the slashes is a non-terminal defined in fig. <>.
<>	The text inside the angle brackets is an identifier.
keyword	A keyword of the metalanguage. Keywords compound of more than one word exist.
Code	Piece of code written in some codification language. It must be enclosed in **{-{** … **}-}** in order to isolate it from the rest of the metalanguage.
//…	A comment that ends with the line.
/* … */	A comment that does not end with the line. It cannot be nested with other comments.

References

[CIM] J.A. Fernández-Madrigal, C. Galindo, J. González, E. Cruz, and A. Cruz, "A Software Engineering Approach for the Development of Heterogeneous Robotic Applications", Robotics and Computer-Integrated Manufacturing (to appear).

[XML] E. R. Harold. XML Bible, 1999, Hungry Minds, Incorporated.

[passive and active replicas] ?

[active objects] ?

[OMG IDL] http://www.omg.org/

[OMG CORBA] Schmidt D. ACE+TAO Corba Homepage, 2005. http://www.cs.wustl.edu/~idt/TAO.html

[acyclic graph]M.R. Trudeau, Introduction to Graph Theory, Dover Publications, 1993.

[BABEL website] Fernandez-Madrigal J.A. The BABEL development system for integrating heterogeneous robotic software. Tech. rep. University of Málaga, 2003. http://www.babel.isa.uma.es/babel/

[migration mechanism] ?

[labview] http://www.ni.com/labview/

[statechart] Harel D. StateCharts. A Visual Formalism for complex Systems. In Science of Computer Programming 1987, vol. 8 1-4.

[ebnf] A summary for the EBNF Notation. Document ISO/IEC 14977 : 1996(E). Accesible online from http://www.cl.cam.ac.uk/~mgk25/iso-ebnf.html

[photoelectric sensor] Diffusive sensor with background suppression. DataSheet RL28-8-H-400-RT-B3B/73c. Pepperl + Fuchs. http://www.am.pepperl-fuchs.com/pdf/documents/rl28-8-h-400-rt-b3b-73c-datasheet.pdf

[conveyor gearmotor and starter] MOVIMOT. http://www.sew-eurodrive.com

[pneumatic diverter] Kuhnke Airbox. http://www.kuhnkeairbox.com/index.php

[SV710 cameras] SIMATIC VS 710. Quick Reference Guide A5E00032597-02 Edition 11/2001

[Simatic S5-100 U] S5-100U. Programmable Controller. System Manual. EWA 4NEB 812 6120-02a

[Profibus-DP] PROFIBUS, System Description. Version Oct. 2002, Order Number 4.002

[CP5431] Siemens Automation & Drives Product Type AS-Interface Master for SIMATIC S5 Order No. 6GK1 243-3SA00

[Profibus-RS-232 gateway] PROFIBUS - RS232 Gateway DSPI_RS, SHAUF http://www.schauf.haan.de

[Ethernet-Profibus-dp] APP-ESP-GTW applicom® GATEway Profibus to Ethernet or Serial. Http://woodhead.com

[Intel] http://www.intel.com

[MS-dos] http://www.microsoft.com

[protected—mode for accessing all the memory] J. Crow. The MS-DOS Memory Environment. ACM SIGICE Bulleting, Vol 21, nº3, Jan 1996.

[Visual C++] D.J. Kruglinski. Inside Visual C++. Microsoft 1997.

[AWL] S5-100U. Programmable Controller. System Manual. EWA 4NEB 812 6120-02a

[WXP] Microsoft homepage (2007). http://www.microsoft.com

[Java] Sun's JAVA homepage (2007). http://www.sun.com/java/

[socket interface] Taylor E. TCP/IP complete. McGraw-Hill Professional, 1998.

[AS-i] http://www.as-interface.net/