Specifying the Concurrent Programming System REGIS in the π-Calculus

Susan Eisenbach, Jeff Kramer and Jeff Magee
Department of Computing
Imperial College of Science, Technology and Medicine
London SW7 2BZ, United Kingdom Phone: +44 71 589 5111 x5063
Fax: +44 71 581 8024
e-mail: se@doc.ic.ac.uk|

October 3, 1994

Abstract

REGIS is a programming system for the development of distributed and parallel programs. REGIS programs consist of three parts. Firstly, there is a configuration part which provides a hierarchical structure of components with dynamic binding. Secondly, there is the actual communication part which provides the interaction and synchronisation required by the system. Finally, there is the computation part providing the component programs written in C++. The subdivision of concurrent programs into the three separate parts of organisation, communication and computation leads to programs that are easy to specify, compile and execute. In order to specify precisely the behaviour of REGIS programs, we translate the organisation and communication into the π-calculus, a formalism for modelling concurrent processes. The π-calculus semantics enables us to deduce behavioural properties of REGIS programs.

1 Introduction

The behaviour of an executing program should not come as a surprise to the writer of that program. Yet with programs that run on parallel and distributed systems, it is notoriously difficult to say with certainty what can be expected. Giving a formal semantics to a programming language enables one to say with certainty what can be expected; without a formal semantics a language is defined by its compiler. Even with the best intentions several compilers for a language will lead to several variants. For any concurrent language designed to be implemented on very different architectures the importance of a formal language specification becomes paramount.

But there is a problem with formality. Unless the world view that the formal system models is the same as that of the programming system it will be easier to execute a program than to prove useful properties about its behaviour. Therefore the underlying model that the specification language supports must be similar to that of the programming language.

The REGIS configuration language [**?**, **?**, **?**] is an interconnection language for the configuration of modules across processors. The REGIS communication system[**?**] enables message transfer for receipt at ports. An important characteristic of the REGIS system is that it enables systems to be configured dynamically by making the addresses of ports first class objects. The REGIS system grew out of Conic which has been used in large scale industrial problems[**?**, **?**].

Several languages such as CSP and CCS that have been devised to specify communicating computational systems. These languages should be suitable for specifying concurrent programming languages[**?**]. However, the modelling of dynamic systems is not straightforward using these languages. Allen and Garlan at Carnegie Mellon[**?**] are developing a promising formalism based on CSP for reasoning about architectural connection. They have been able to use their formalism to be able to prove properties about termination and consistency of programs. So far it ignores hierarchy and dynamic connections.

Milner's recent system the π-calculus [**?**, **?**] is designed to model concurrent computation consisting of processes which interact and whose configuration is changing. It does this by viewing a system as a collection of independent processes which may share communication links or bindings with other processes. Links have names. These names (or addresses) are the fundamental building blocks of the π-calculus. REGIS's port addresses are like these names, helping to make the π-calculus a good system for describing the communication. To date though, both the examples undertaken in the π-calculus and the languages specified in it have not been substantial[**?**]. So being able to define REGIS configuration and communication in π-calculus is also a demonstration that the calculus has the expressive power required for solving real problems.

In this paper brief descriptions of the REGIS system and the π-calculus are given. An example to demonstrate the expressive power of REGIS and the π-calculus is developed. This is followed by a formal semantics of REGIS in the calculus. The paper concludes with what can be deduced about the behaviour of REGIS programs and a discussion of the value of specifying a concurrent programming system.

2 REGIS

2.1 The configuration language

The REGIS configuration language[**?**, **?**, **?**] is intended for the description of both static and dynamic configurations of processes. Composite components are written in the configuration language itself and primitive components are written in C++. A grammar of the configuration language is given in Figure **??**.

The basic unit of the configuration language is a component or process. A REGIS configuration component consists of a list of declarations. Actual instances of components are declared using the **inst** keyword. The **inst** declaration actually causes a process to be created. The interface of a component with its environment is a vector of names that are **provide**d by the process, and another vector of names that are **require**d by the process. These names

$$
\begin{array}{rcl}
\textit{program} & = & \textit{component}^* \\
\textit{component} & = & \textbf{component}\ \textit{id}(\textit{parameters}) : \textit{instance-type}\ \textit{body} \\
\textit{body} & = & \{\ \ \{\ \textit{decl}^*\ \textit{action}^*\ \}\ \} \\
 & | & \textit{primitive-component} \\
\textit{type} & = & \textit{instance-type} \\
 & | & \textit{component-type} \\
 & | & \textbf{array}\ [\textit{num}_1..\textit{num}_2]\ \textbf{of}\ \textit{type} \\
 & | & \textit{primitive-type} \\
\textit{instance-type} & = & \textbf{require}\ \textit{decl}^*; \textbf{provide}\ \textit{decl}^*\ ; \\
\textit{component-type} & = & \textbf{component}\ (\textit{parameters})\ \textit{instance-type} \\
\textit{decl} & = & \textit{id} : \textit{type} \\
\textit{action} & = & \textbf{inst}\ \textit{instance-id}\ \texttt{:=}\ \textit{component-id}(\textit{arguments}) \\
 & | & \textbf{bind}\ \textit{service-name}_1\ \texttt{--}\ \textit{service-name}_2 \\
 & | & \textbf{when}\ \textit{expr}\ \{\ \textit{action}^*\ \} \\
 & | & \textbf{forall}\ \textit{id} : \textit{expr}_1\ \texttt{..}\ \textit{expr}_2\ \{\ \textit{action}^*\ \} \\
\textit{service-name} & = & \textit{id} \\
 & | & \textit{instance-id.id}
\end{array}
$$

Figure 1: A grammar for the configuration language

may be of any type; the REGIS configuration language does not place any restrictions on the types (but it does ensure that program construction is type secure). Although the configuration language does not place any restriction on provisions and requirements, in the examples used here they will refer to communication ports. The names of provisions and requirements may be referred to in the program defining the process. In a primitive component, written in C++, the provided names are supplied with values, while required names are unresolved references. Any reference to an undefined name by a process will cause the process to block until a value is assigned.

To actually bind two components together to enable the transference of data a **bind** declaration is used. This is the core statement of the configuration language. bind $\mathrm{id}_1.\textit{requirement} - -\textit{id}_2.\textit{provision}$ assigns the provided value of one instance to the required name of another. The **bind** statement may also refer to the provisions and requirements of the process being defined. The following combinations are permitted: bind instance.requirement – instance.provision

bind provision – instance.provision

bind instance.requirement – requirement

bind provision – requirement However, no variable may be bound twice.

In the configuration program, all actions may be performed concurrently. Thus the name on the right side may not have a value when the **bind** statement is performed. The semantics must be defined in such a way that the order in which the bindings are performed not restricted.

In the configuration program, all actions may be performed concurrently. REGIS also enables a limited amount of control structuring of its declarations. **When** declarations are guards that enable one or more declarations to be declared optionally. It is not uncommon to want to perform the same declaration repetitively. **Forall** declarations enable this to take place.

2.2 The communication

template ¡ class T ¿ class port : public portbase public: void in(T msg); // receive message of type T into msg int inv(T msg[], int n); // receive inv elements of vector (maximum n)
void out(T msg); // synchronous send msg of type T void outv(T msg[],int n); // synchronous send of n elements of msg
void send(T msg); // asynchronous send msg of type T void sendv(T msg[],int n); // asynchronous send of n elements of msg ;

Figure 2: Template class for REGIS Port Objects

The communication model of REGIS is implemented as a set of C++ template classes that supports component interaction through message-passing. A communication object is contained within the component which provides the object and is remotely referenced by the component which requires it. Figure **??** is the C++ description of a REGIS port communication object.

Port objects are really queues of messages of a particular type *T*. Messages may be queued to a port object (*out, send*) and removed from a port object (*in*). A communication object named *input* for integer messages would be declared in C++ as: port ¡int¿ input; The operation: *input.send(3)* would queue an integer message with the value *3* to the port input. This send is asynchronous in the sense that it does not block its calling process. The synchronous operation *out* blocks its calling process until the message has been received. Variable length messages may be transferred through ports using the vector primitives (*inv, outv, sendv*). The operations on ports are implemented using the underlying class *portbase* which supplies untyped *send* and *receive* operations. In addition, *portbase* supplies methods common to all ports. These methods are used to selectively wait on a set of ports and to determine whether messages are queued to a port.

The port operations described so far are only available to invoking processes which are co-located with the port object. That is, they are resident on the same processor and in the same address space such that they can either name a port directly or use a pointer to it. This is of limited use in a distributed memory environment. Consequently, the implementer of communication objects in the REGIS framework must describe not only the object class but also a class which can be used to access that communication object remotely. These remote access classes are by convention named by adding the suffix *ref* to the name of the communication object class it is providing access to. REGIS provides the template class *portref* for remote access to port objects.