A Rigorous Method for the Constructive Design of Parallel and Distributed Programs

George R. R. Justo
Centre for Parallel Computing
University of Westminster, London
e-mail: justog@wmin.ac.uk,

Abstract
Parallel and distributed systems engineers are always looking for a way to speed-up their programs. They sometimes forget that well-structured programs are more flexible, and therefore easier to modify or restructure in order to improve performance or to map onto a particular architecture.

This paper illustrates a systematic way of designing well-structured parallel and distributed programs. The method is based on SASD [1], one of the most popular methods for the analysis and design of sequential systems, and CSP [2], a formalism for specifying the behaviour of communicating systems. The influence of SASD is evident in the way diagrams are used during the various phases of the development. CSP allows us to formally verify and transform the programs.

The main feature of our method is the ability to reuse behavioural specifications, the way the components synchronise and communicate, and provide rules to verify and transform the design structure.

1 Introduction
Software engineering techniques and methods are essential in the development of any piece of software. Terms like modularity, systematic analysis and design, and more recently object-oriented development have become common. In addition an important practice that has been proved essential is the need for systematic analysis and design before the system is implemented.

Unfortunately, most parallel and distributed systems engineers seem to ignore these lessons. They have been haunted by performance improvement and how to obtain it at any cost. The main consequence may not come now but will eventually appear when changes have to be made in the system, especially when these systems have to be modified (the old problem of maintenance!), especially when we have to reconfigure our parallel programs to a different architecture or topology. The problem becomes potentially yet more serious because of the possible widespread use of parallel machines provided by clusters of workstations and networks of PC's.

We do not ignore the importance of performance in the development of parallel and distributed systems (PDS) but the performance issues should be integrated in the development process, and could ideally be adjusted by clever compilers or run-time systems. Unfortunately, we do not believe in clever parallel compilers that can take a sequential program or system and generate efficient and well-designed PDS. This solution is very appealing but only works for certain algorithms and particular languages. It cannot be applied to general purpose applications. The PDS engineer, therefore, must play an important role in analysing and decomposing the system into parts that can be executed in parallel. The compiler or run-time system can assist the engineer in load balancing the parallel parts on the target hardware architecture.

The objective of this paper is to present a method for the designing of PDS. We recognise the importance of the structural description of PDS because this information can help in distributing the systems' components on the parallel architecture. The importance of the structural description has been used in other methods (for example [3]) but they usually fail to integrate the structural description with the behavioural description of the components which is essential in the development of PDS. The components are usually seen as "black boxes" and to observe their behaviour we must examine their internal structure. In addition, the internal structure (behaviour) is usually specified in terms of a finite state machine (FSM) and there are many problems associated with the use of FSM's [4], for example, lack of support for hierarchical decomposition and state explosion.

Another approach is the use of object-oriented pictures [5]. The graphical language allows us to describe the components' behaviour in great detail (the description resembles the implementation) but it does not allow the same degree of formal analysis of CSP [2] or CCS [6].

Despite the arguments against formal methods for not being practical, they play an important role in producing reliable systems. A solution to the problem of presenting formal methods is suggested by [7] - algebraic rules that relate the objects that we (systems engineers) are used to, programs. The idea is to define simple laws or design rules that allow us to manipulate the programs directly without having to be aware of how the laws are proved or used by the mathematicians.

In summary, we will describe a method based upon the construction and manipulation of the structural
description of systems. But this description will also include the behavioural description of the system. We will use the notion of template which specifies pre-defined reusable process behaviours. Based upon the behavioural description and the templates, design rules are provided to allow the systems engineer to manipulate (verify and modify) the system description without having to develop complicated proofs.

2 Analysis

The specification of the interactions amongst entities is essential in modelling concurrent (communicating) systems, and should be considered at all levels of the development [8]. The identification of the concurrent aspects of a system during the requirement analysis is important in guaranteeing that the program exhibits the parallelism of the problem domain.

Structured Analysis (SA) is one of the most popular techniques used in the specification of requirements. The main principle is that the structure of the specification will be directly used to model the solution (design). The central component of SA is the dataflow diagram (DFD) [9] which describes the system as a set of processes that modify or change data that pass along the dataflows.

Unfortunately, there are several arguments against the use of DFD in the specification of communicating systems. The main one is the lack of expressiveness to specify the interactions between concurrent components. The DFD only specifies the flow of messages between components but it does not say anything about the synchronisation or the conditions for interaction. Another aspect of SA is that it produces "function-oriented" designs.

The decomposition of a system in terms of "objects" seems more appropriate for modelling communicating systems because objects usually denote "active" entities that interact with other entities. This helps in identifying the natural concurrent activities of the problem domain and modelling it in the design. An important property of object-oriented approaches is the use of "inheritance" as a powerful mechanism to specify hierarchically the structure of the system. However, this kind of structure is not relevant to the construction of PDS. In fact, problems can arise later in the development because objects of related classes may shared data (for example, objects of the same class may share static variables) and we won't be able to allocated them to distinct nodes [10].

During the analysis, our approach uses a diagram similar to DFDs but we model objects and their interactions, instead of functional processes. Our method is object-based and not object-oriented [11]. This means that a hierarchy only denotes the decomposition relationships between components and not inheritance relationships. Also, we are only interested in active objects [10]. In our case, the active objects perform autonomous actions and interact with other objects. Passive objects are those on which computations are carried out. We consider them data types and they become part of particular active objects.

The result of this phase is a collection of objects and the relationships (message flow) amongst them - that is, an interaction diagram (graph) (ID). The ID is similar to the collaboration graph [12] and the message diagram [13]. The former shows the relationships between collaborators - that is, the connections between the various classes. The latter can be seen as a detailed version of the collaboration graph where the arrows represent the actual messages that can be exchanged amongst the objects belonging to the various classes.

The behavioural specification of the components is not included in the ID diagram. It will be included in a more detailed diagram presented in Section 3.

2.1 An example

To illustrate the use of our method we will develop a Hospital System. The objective of the system is to perform the main tasks of a patient treatment automatically. The doctor receives information about the patient's symptoms and the nurse's report about the patient progress. The doctor is responsible for stating the kind of treatment that must be applied to the patient. The nurse monitors the patient's condition and applies the treatment prescribed. The nurse also requests the pharmacist to deliver the medicine required for the patient's treatment.

Using an object-oriented analysis approach, we identify the principal active objects of the system: Nurse, Patient, Doctor and Pharmacist. Note that we do not consider objects like medicine and prescription, since they do not define "active" objects. In the informal description of the problem, there is no reference about actions performed by medicine or prescription. In this case, they represent just data types.

The interactions between two objects are defined when one object offers some action (service) requested by the other object. In the graph, a directed edge connects the object that performs the action to the one that requests it. The ID for the Hospital System is presented in Figure 1.

![Figure 1: ID for the Hospital System.](image-url)
3 Design

The first phase according to SASD [1] (logical design) corresponds to the transformation of the DFD defined during the analysis into a design, that is, a set of related (functional) modules. This shows the relationship between the requirements specification and the design. The systems engineer can then validate its solution against the specification. However, this first design is not in general the final solution because the requirement specification is not supposed to include any constraint information about the environment in which the system will execute. This information is only used during the design.

To specify the functionality of each "process", SASD uses the notion of mini-specification [9]. A mini-specification describes what the process does or the kind of change the "process" performs on the data. Certainly, this kind of specification does not describe the interaction of the "process" with its external environment as it is required for PDS. This information is more precisely described by a behavioural specification.

Process algebras [14] represent a simple, elegant and precise approach to specifying behavioural specification of processes. We propose the use of CSP [2] that provides three different views of the specification: denotational (refusal sets [2]), operational (transition systems [15]) and transformational (algebraic laws [16]). These three views enable us to manipulate the specifications in different ways.

One of the limitations of DFDs in describing communicating systems is that they do not include the behavioural aspects in the graph. The behavioural specification is usually represented through event-driven diagrams [1] or FSM’s. This approach usually describes how the processes react to events (messages) they receive and which events they generate. We have already discussed the problems of FSM’s. Statecharts [4] present solutions for the limitations of FSM’s. In particular, they are structured and hierarchical. However, the arrows in a statechart (or FSM) always represent state transitions and not message flow, and the blobs (or nodes) always denote states and not processes or objects.

SCOOP-3TM [5] presents a better solution for the problem as the arrows can represent data flow, message flow and control flow, and the nodes denote objects, structured control nodes and data stores. However, the graphs present too much information and may become very confusing.

In our graphs, the behavioural information can be described by special nodes that denote different templates. A template [17, 18] defines a particular kind of behaviour, namely, a reusable pattern of communication and synchronisation between the process and its environment. The behaviour of each template is formally defined in CSP. For example, an S/R (IO-PAR) template sends and receives messages in parallel. More formally, if we assume the alphabet of an S/R template P to be \( \alpha P \), then the behaviour of P is defined by the following traces [2],

\[
\text{traces}(P) = \text{INTERLEAVES}(\alpha P) \\
\cup (\text{INTERLEAVES}(\alpha P) \setminus \text{traces}(P))
\]

where the function \( \text{INTERLEAVES} \) computes permutations of events of the alphabet, the function \( \text{INTERLEAVES}^\ast \) applies \( \text{INTERLEAVES} \) to the powerset of the process’ alphabet, and \( \setminus \) denotes the CSP catenation of traces. The definition above determines that the traces of P are described by all interleavings of events in the alphabet.

In our method, the systems engineer should choose a template that most approximates the expected behaviour of each process. The design is like playing with "lego" blocks in order to build the system that behaves in the way we expect. If no template appears to satisfy the expected behaviour, the user should postpone the specification of that process. The importance of identifying templates for every process is that the design rules are based on templates. However, even if the template does not match directly the process’s behaviour, it can be used as a first approximation or prototype that can be modified or refined later.

The level 1 design is very similar to the ID. The objects are transformed into processes (boxes) that will be executed concurrently, and the interactions are transformed into messages between processes (message edges). The result is a graph called message-flow diagram (MFD).

3.1 The design of the hospital system

Figure 2 shows the level 1 MFD for the Hospital System before the behavioural specification of the processes is included.

The next step of the design is more important. It consists of defining the behaviour of the processes or more precisely identifying templates that describe the behaviour of the processes. We have presented several templates in [19, 18].

We can either identify a template to each process individually and modify it if it fails in the verification, or we can choose the templates taking in consideration the goal of deadlock-freedom.

To illustrate how the method can produce various alternative solutions, we consider the first option in this section, and we will discuss the second option in the next section.

We will start by observing the Patient 2 process. It first sends two messages (Patient_Symptoms and Patient.Condition), and then waits for Applied_Treatment. This corresponds to the behaviour of a (parallel) S-R (IO-SEQ) template. A parallel S-R sends a set of messages in parallel, performs some computation and finally receives some messages in parallel.

Process Doctor 3 receives two messages (Patient_Symptoms and Patient_Report), and with these messages it generates the Prescribed_Treatment. This is the behaviour of a (parallel) R-S (IO-SEQ)
template which receives a set of messages in parallel, performs some computation and sends some messages in parallel. In the case of process Doctor 3, it sends only one message.

Process Pharmacist 4 behaves like process Doctor 3, that is, it receives a Request_Medicine message, and then sends Deliver_Medicine with the prescribed medicine or an error. The nature of the message however is not important at this stage, we are only interested in the interactions.

We will specify the behaviour of process Nurse 1 in the next section. However, we can now present the refined MFD. The idea is to represent each process with the icon that denotes the template, and only by observing the graph can we deduce the behaviour of the process. The refined MFD of the Hospital System is presented in Figure 3.

3.2 Design verification

Graphical representation has been widely used in software development as a main way of expressing and simplifying the complexity of software systems. Graphical notations have been proposed for almost all phases of the development, from requirement analysis, for example Extended Petri Nets [20], to program execution animation, for example Balsa-II [21].

What makes a graphical representation really useful is its analytical and consistency characteristics [1]. In terms of a graphical notation that represents the behavioural aspects of a process, the notation should permit analysis of the behavioural properties of the process. The most important behavioural property of PDS is that it guarantees that all interactions (message passing and synchronisation) occur correctly - that is, deadlock-freedom.

Despite being an essential property of PDS, deadlock-freedom is not easy to verify. The reason being the large number of states that must usually be analysed to guarantee that a system never enters a deadlocked state, where all processes block trying to communicate. Some techniques [22] have been proposed aiming to reduce the number of states that must be analysed but in general this still means that many states must be checked.

An important objective of a design method is to assist in improving the quality. It is obvious that correctness is an essential quality. Our method provides simple rules that can be applied during the design to guarantee that the system is deadlock-free.

General rules are not possible but when we define our system from pre-defined behavioural structures (our templates), it is possible. This is one of the main advantages of a "lego-like" development.

The rules define which kind of combination of templates may or not deadlock. More precisely, the rules establish topologies (connections or interactions) of templates that are safe or not. So, the rules can be used to guide us in defining the topology. If a topology is not safe, we have to change the connections between the processes or the types of templates associated with the processes in order to make the design safe.

The rules also provide a systematic approach that the systems engineer can apply to experiment on different alternative designs.

Even when every process' behaviour has not been specified, we can still apply the rules to sections of the MFD. This can help in defining context properties that will be used to specify the behaviour of unspecified processes to guarantee that they will not cause deadlock.

3.3 Verifying the hospital system

The main strategy for identifying deadlock is to observe the dependencies between the processes. The fatal dependency is when process A tries to send a message to process B that cannot receive that message because it is also trying to send a message to process A. This means that there is a cycle in the interaction between the two processes. But not every cycle causes deadlock. Therefore, the main objective of the verification is to check which cycles can cause deadlock and modify them.

In Figure 3, we can see that there are a lot of edges from and to Nurse 1 process. During the identification of a template with that process, we should consider not only the behaviour of the process but also the possibility of using a template that helps to avoid deadlock. Let's observe three possible behaviours for Nurse 1 process. This will show how our method can help the PDS engineer to experiment with alternative design solutions.

Process Nurse 1 could behave like a (parallel) R-S template which could wait for the Prescribed_Treatment, the PatientCondition and an initial (empty) Deliver_Medicine. After that, it could send Applied_Treatment, Patient_Report and Request_Medicine. Unfortunately, by the verification rules (more details in [18, 19, 24]), a cycle that includes two (parallel) R-S templates always deadlock. Since, there would be a cycle between processes Nurse 1 and Doctor 3 which would be R-S templates, there would be a deadlock.
Process Nurse 1 could behave like a (parallel) S-R template by sending an initial message to the other processes specifying that it has just stated working, and then it receives the Prescribed treatment, the Patient Condition and an empty delivery from Pharmacist 4 process. By the verification rules, a cycle of (parallel) R-S and (parallel) S-R templates does not deadlock. When we verify each cycle in the MFD, we observe that it includes Nurse 1 process. So every cycle is deadlock-free.

Another strategy is to consider any process that has many interactions as S//R (IO-PAR) template. This process is the one that sends and receives all its messages in parallel. It reduces the chances of deadlock, since it avoids dependencies between the sends and receives. According to the verification rules, any cycle that includes an S//R (IO-PAR) template is deadlock-free. Another advantage of using an S//R (IO-PAR) template is how easy it is to be decomposed. This will be shown in the next section. A process that has many interactions can be easily decomposed into subprocesses that performs part of the interactions.

Figure 4 shows the refined MFD of the Hospital System, assuming that Nurse 1 process behaves like a S//R (IO-PAR) template.

### 3.4 Design refinement

The first design derived from the analysis does not usually represent the final solution for the system, as stated before. We have particularly to examine each process (module) in terms of various properties. A major aspect is the relationship of the internal components of the process (the degree of cohesion). But more important for PDS is the degree of concurrency or parallelism. To measure the optimal degree of parallelism of a design is in general part of the creative part of the design. Also, it requires details of the environment where the system will execute. So, it should not be included at this stage of the development.

There are, however, two important points worth considering. First, as in the case of traditional software design, each process should have a clear and well-defined function in order to improve modularity. Even in an object-oriented approach, sub-processes should correspond to the methods (functions) of the object. Secondly, “parallel slackness” [23] is an important property for PDS. This means that there should always be more processes than processors to guarantee that no processor will be idle because of communication latency. In terms of design, it signifies the decomposition of processes into smaller processes that are more modular and that increases the “parallel slackness” of the system.

When we decompose the system two aspects must be considered. First, the refined version of the system should behave like the higher level version. Secondly, as the number of processes and interactions increases the chance of deadlock also increases.

Our method provides some rules that assist the systems engineer in solving these two problems [24]. Each rule relates a template with a behavioural equivalent configuration of templates. Since they are behavioural equivalent, they have the same behavioural properties. This is what we expect of the refinement. Also, each template defines a single process without internal communication that is free of (internal) deadlock. Therefore, any process or configuration of processes that is behavioural equivalent to that template is also deadlock-free.

### 3.5 The refinement of the hospital system

The Nurse 1 process is a prime process for refinement (decomposition) because of the number of interactions (functions) it performs.

If we follow an object-oriented approach, the process could be decomposed into its functions or services. However, many of the interactions represent only requests for services or data produced by other processes. So, the decomposition should also take into consideration the interactions. In the same way that we aim at increasing cohesion and reducing coupling, we should increase “interaction cohesion” and reduce...
"interaction coupling". We follow a similar technique to functional cohesion since each interaction is related to some function of the system. In this sense, we group all interactions related to each function that may include the inputs (required services) and results (produced services).

The first decomposition of Nurse 1 is the separation of inputs and outputs. Nurse 1 collects some data and generates reports and requests. This decomposition is illustrated in Figure 5(a).

A further decomposition consists of separating the data collection function where the data require different consistency checks and are used in different ways. Similarly, the outputs are decomposed into a function that generates the patient’s report, one that requests medicine from pharmacist and one that applies the treatment to the patient. This decomposition is shown in Figure 5(b).

We have to prove that the refined version of Nurse 1 process behaves like the original process. In our method, we use refinement rules that guide us in defining the behaviour of each sub-process in the refined version. Each refinement rule associates a template with a configuration of templates that has the same behaviour (modulo the environment – precedence equivalence [24]). In this example, we apply two rules. Rule 2 says that two templates, an R-S (IO-SEQ) and an S//R (IO-PAR), where the R-S template must only send messages to the S//R template is equivalent (precedence equivalence) to an S//R template. The second rule, Rule 3, is the reverse – that is, the R-S template must only receive messages from the S//R template. So, we can relate process Store_Compute 1.1 with an S//R template, and all the other processes are R-S templates as Figure 6 shows.

The proof of equivalence between Nurse 1 process and its refined version is very simple. We can apply Rule 2 to merge each process, Collect_from_Doctor 1.2, Collect_from_Patient 1.3 and Collect_from_Pharmacist 1.4 with process Store_Compute 1.1, resulting into a single (equivalent) S//R template. Similarly, we can apply Rule 3 to merge Store_Compute 1.1 with processes Report_to_Doctor 1.5, Apply_Treatment 1.6 and Request_to_Pharmacist 1.7, resulting into a single S//R template that behaves like Nurse 1 process.

Notice that Rule 2 and Rule 3 could have also been used in the refined version of Nurse 1 process presented in Figure 5(a).

As the method is graphics-based, all design rules are also represented graphically. The graphical representation of some refinement rules is presented in [25]. For example, Figure 7 shows the graphical representation of Rule 2 and Rule 3 where the shaded area in the R-S template denotes that it only sends internal messages to the R//S template in Rule 2 or only receives internal messages from the R//S template in Rule 3. The dotted lines denote interactions that may occur but are not a condition for application of the rule.

3.6 Top-down vs. bottom-up design

There are many arguments for top-down design, especially as it embodies an ancient human principle of solving problems – divide and conquer. However, some experience has shown that many systems are in fact developed in a bottom-up manner [1]. One of the reasons for this is that many systems are developed from existing systems where a lot of details are known or during the analysis the information is collected from the bottom of the organisation from people directly involved with the system.

The important fact is that bottom-up development can be the natural development approach to many systems. Also important is the issue of reusability, especially in object-oriented approaches.

The reason that we have presented our method as a top-down approach is more historical than technical. Our method can be easily applied to a bottom-up development also. The refinement rules we have presented denote equivalence between a template and a configuration of templates. We can also apply the
Figure 5: Level 2 MFD of Nurse 1 process.

Figure 6: Level 2 MFD of the Hospital System.
rules during the synthesis (abstraction) – decomposition in reverse. An already defined configuration of templates can be decomposed into sub-configurations and the rules can be applied to each sub-configuration to substitute it for a single template.

In a similar way, we have applied the rules to merge processes during the implementation in order to reduce the number of processes. This is used as an optimisation technique that we call serialisation [26], the reverse of parallelisation.

3.7 Process behaviour elaboration

Up to now, we have specified the behaviour of the processes in terms of CSP traces or refusals [2]. This kind of specification is very abstract and there is a big gap between the implementation and the specification. The method provides more concrete specification of the processes’ behaviour in CSP. It is clear that the systems engineer can derive his/her own CSP specification from the traces or refusals. The CSP specification only describes the behaviour of the process, and the process’ functionality is assumed to be orthogonal to the behavioural specification – that is, the behavioural specification has gaps (compute processes) that are filled in with the process’ functional aspects. The compute processes are supposed to have no external communication and must terminate. As an example the traces of the S//R (IO-PAR) template presented in equation 1 can be elaborated as the following CSP process:

\[
P = \text{compute}' \quad \quad P' = \text{compute}''
\]

\[
( | | | a(i) \in aP(a(i) \rightarrow \text{compute}(i)))
\]

where process P first behaves like process compute', and then enters a loop that repeats process P'. Process P' first behaves like process compute'' and continues by behaving like a composed concurrent process where each sub-process i performs an action a(i) of the aP before a process compute(i).

We also define implementation templates from the CSP specification. Unfortunately, the proof of satisfaction of the implementation cannot be totally formalised for most languages that do not present formal semantics. But for some languages that are formally defined, like occam [27], we can develop the proof. We present below a template in C and PVM [28] that implements the S//R template: for simplification we omit the declarations and some PVM functions:

```c
main () /** S_R-template **/ {
    /* variable declarations */
    /* enroll in pvm */
    mytid = pvm_mytid();
    /* initialisation */
    compute:
    while ( true ) {
        /* loop computation */
        compute_;
        /* communication */
        for ( i=0; i < nsends; i++ ) {
            /* non-blocking send */
            pvm_send ( a[i], type[i] );
        }
        while ( waiting ) {
            for ( i=0; i < nactions; i++ ) {
                if ( waiting[i] )
                    /* non-blocking receive */
                    if ( pvm_nrecv ( a[i], type[i])>0 )
                        compute(i);
                    /* send acknowledgement signal */
                    ack_if Necessary(i);
                waiting[i] := false;
            }
        }
    }
}
```
We implemented the blocking send in PVM by allowing the process to send the message but it can only continue its computation after receiving an acknowledgment (signal P) from the receiver. The concurrency of the actions in CSP has been implemented by non-blocking receives.

4 Conclusions and future work

As we have stressed throughout the paper, our method is suitable for the development of message passing systems. The method is based upon a model with the following properties: modularity, component independence, explicit structural description and interaction via message passing. So, the method presents a systematic way to obtain and explore these properties. The systems engineer can experiment with different designs in order to find one that presents the 'best' use of the properties above.

The method has been successfully applied to the development of parallel programs implemented in oc-cam, programs implemented in CL [29] and programs implemented in C with PVM.

The concept of object idioms [5] has been widely used as a method of reusing analysis and design components. The concept of template that we use in our method follows the same principle of reusability but we reuse the proofs of deadlock-freedom and refinement (synthesis). This is an effective way of improving reliability and also reducing the systems development effort.

Unfortunately, reusability has its limitations. The templates cannot be used to model any kind of systems but their applicability is quite extensive. It requires the creativity of the systems engineer to model a system using the templates but system design is a creative process!

Another important aspect is the use of the templates for prototyping or first versions of the design. It may help the systems engineer to understand the properties (behaviour) of the system.

It was not an objective to produce a complete set of templates. The idea was to define small sets of templates that could be used in particular applications. The reason being the practical aspect of using the templates, because with too many templates it would be difficult to identify a template with a particular process. We have concentrated on classes of problems and we have defined suitable templates, for example, for regular and client-server configurations. We have also shown in [19] that it is possible to combine different classes of templates. At the moment we are defining new templates for vision applications and discrete event simulation.

5 Acknowledgement

I gratefully acknowledge the reviewers for their comments and suggestions on improving this paper.

References


