Beyond Definition/ Use: Architectural Interconnection*

Robert Allen David Garlan

Computer Science Department
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15206-3891

Workshop on Interface Definition Languages, Portland, Oregon, January 20, 1994.

Abstract

Large software systems require decompositional mechanisms in order to make them tractable. Traditionally, MILs and IDLs have played this role by providing notations based on definition/use bindings. In this paper we argue that current MIL/IDLs based on definition/use have some serious drawbacks. A significant problem is that they fail to distinguish between "implementation" and "interaction" relationships between modules. We propose an alternative model in which components interact along well-defined lines of communication – or connectors. Connectors are defined as protocols that capture the expected patterns of communication between modules. We show how this leads to a scheme that is much more expressive for architectural relationships, that allows the formal definition of module interaction, and that supports its own form of automated checks and formal reasoning.

1 Introduction

Large software systems require decompositional mechanisms in order to make them tractable. By breaking a system into pieces it becomes possible to reason about overall properties by understanding the properties of each of the parts. Traditionally, Module Interconnection Languages (MILs) and Interface Definition Languages (IDLs) have played this role by providing notations for describing (a) computational units with well-defined interfaces, and (b) compositional mechanisms for gluing the pieces together.

A key issue in design of a MIL/IDL is the nature of that glue. Currently the predominant form of composition is based on definition/use bindings [13]. In this model each module *defines* or *provides* a set of facilities that are available to other modules, and *uses* or *requires* facilities provided by other modules. The purpose of the glue is to resolve the definition/use relationships by indicating for each use of a facility where its corresponding definition is provided.

*This research was sponsored by the National Science Foundation under Grant Number CCR-9357792, by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under grant number F33615-93-1-1330, and by Siemens Corporate Research. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of Wright Laboratory, the U.S. Government, or Siemens Corporation. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon. This manuscript is submitted for publication with the understanding that the U. S. Government is authorized to reproduce and distribute reprints for Governmental purposes.

This scheme has a number of benefits. It maps well to current programming languages, since the kinds of facilities that are used or defined can be chosen to be precisely those that the underlying programming language supports. (Typically these facilities support procedure call and data sharing.) It is good for the compiler, since name resolution is an integral part of producing an executable system. It supports both automated checks (e.g., type checking) and formal reasoning (e.g., in terms of pre- and post-conditions).

Indeed, the benefits are so transparent that few question the basic tenets of the approach. In this paper, however, we argue that current MIL/IDLs based on definition/use have some serious drawbacks. As we will show, a significant problem is that they fail to distinguish between "implementation" and "interaction" relationships between modules. The former are useful (and necessary) for understanding how one module is built out of the facilities of others. However, the latter are needed to express architectural relationships – such as the nature of the communication between computational components.

To make this point concrete we propose an alternative kind of glue. In this model computational *components* interact with other components along well-defined lines of communication – or *connectors*. Connectors are defined as protocols that capture the expected patterns of communication between modules. We show how this leads to a scheme that is much more expressive for architectural relationships, that allows the formal definition of module interaction, and that supports its own form of automated checks and formal reasoning.

2 Implementation versus Interaction

At the level of system design one important relationship between modules is that of "implements": a given module is defined in terms of facilities provided by other modules. For example, one module might import a string package that it uses to implement an internal data representation.

But this is not the only important kind of relationship. When people design systems they typically provide an architectural description consisting of a set of computational components and set of inter-component connections that indicate the interactions between those components. Often these descriptions are expressed informally as box and line diagrams, and the interaction relationships are described idiomatically with phrases such as "client-server interaction", "pipe and filter organization", or "event-broadcast communication". In these descriptions the components are treated as independent entities that may interact with each other in complex ways [5, 11].

The distinction between a description of a system based on "implements" relationships and one based on "interacts" relationships is important for three reasons. First, the kinds of description involve different ways of reasoning about the system. In the case of implementation relationships, reasoning typically proceeds hierarchically: the correctness of one module depends on the correctness of the modules that it uses. In the case of interaction relationships, the components (or modules) are logically independent of each other: the correctness of each module is independent of the correctness of other modules with which it interacts. Of course, the aggregate system behavior depends on the behavior of its constituent modules and the way that they interact.

Second, the two kinds of relationship have different requirements for abstraction. In the case of implementation relationships it is usually sufficient to adopt the primitives of the underlying programming language – e.g., procedure call and data sharing. In contrast, interaction relationships often involves abstractions not directly provided by programming languages: pipes, event broadcast, client-server protocols, etc. Whereas the implementation relationship is concerned with how a component achieves its computation, the interaction relationship is used to understand how that computation is combined with others in the overall system. Hence, the abstractions used reflect diverse and potentially complex patterns of communication, or protocols.

Third, they involve different requirements for compatibility checking. In the case of implementation

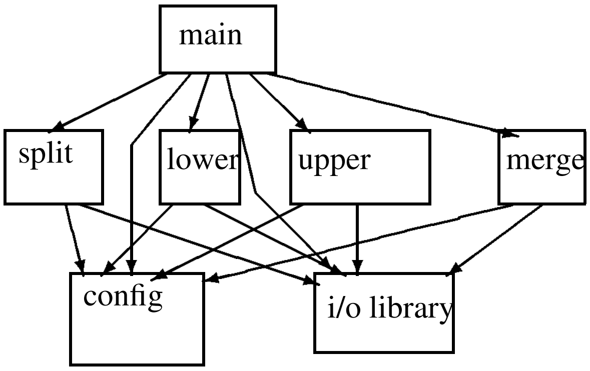

Figure 1: An Implementation Description.

relationships, type checking is used to determine if a use of a facility matches its definition. (Of course, this can be enhanced with reasoning about specifications [6].) In the case of interaction relationships we are more interested in whether protocols of communication are respected. For example, does the reader of a pipe try to read beyond the end-of-input marker; or is the server initialized before a client makes a request of it.

Current MILs are well suited to describing systems in terms of their definition/use dependencies, and hence their implementation relationships. However, they are not adequate for describing architectural interactions. In the remaining sections we illustrate this point and provide an alternative model that is better suited to describing interaction relationships.

3 Example

To illustrate the distinctions outlined above consider a simple system, *Capitalize*, that transforms a stream of characters by capitalizing alternate characters while reducing the others to lowercase.

Let us assume that the system is designed as a pipe-filter system that splits the input stream (using the filter *split*), manipulates each resulting substream separately (using filters *upper* and *lower*) and then remerges the substreams (using *merge*). In a typical implementation of this design we would likely find a decomposition such as the one illustrated in figure 1. It consists of a set-up routine (*main*), a configuration module (*config*), input/output libraries, as well as modules for accomplishing the desired transformations. The set-up routine depends on all of the other modules, since it must coordinate the transformations and do the necessary hooking up of the streams. Each of the filters uses the configuration module to locate its inputs and outputs, and the i/o library to read and write data.

While useful, this diagram fails to capture the architectural composition of the system. It indicates what modules are present in the system, and to what modules their implementations refer.[1] But it fails to capture the overall system design, or architecture. The pipes in this pipe-filter system are not shown.

An alternative representation of the system is shown in figure 2. In contrast to the previous description, the interaction relationships (i.e., the pipes) are highlighted while implementation dependencies are suppressed. Of course, for the picture to have any meaning at all there must be a shared understanding of the meaning of the boxes and lines as filters and pipes.

[1] In this simple example we use the term “modules” loosely to represent separable coding units.

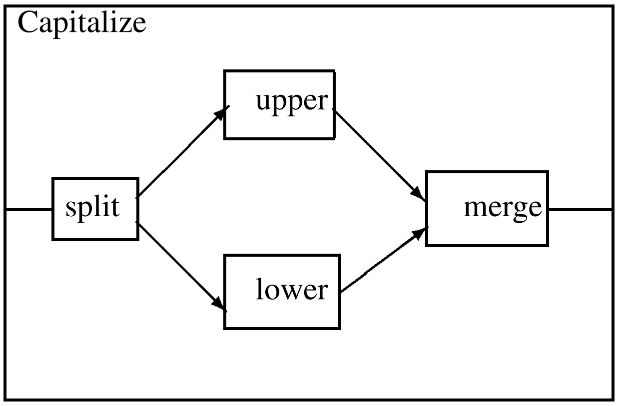

Figure 2: An Architectural Description.

This second description clearly highlights the architectural design and suggests that in order to understand a system it is important to express not only the definition/use dependency relations between implementation "modules," but also to reflect directly the abstract interactions that result in the effective composition of independent components. In particular, to understand and reason about *Capitalize* it is at least as important to know that the output of *upper* is delivered to *merge* as it is to know that it is invoked by *main* and uses *i/o library*.

This example illustrates the three distinctions outlined above. In terms of reasoning about the system, the first description shows a hierarchical decomposition of the system functionality. A specification of the module *main* will be the specification of the behavior of the system as a whole. The first description indicates that the implementation of *main* refers to declarations in the other modules. To demonstrate the correctness of this module, the other modules' specifications must be used, and their correctness shown. In the second description the four filters are represented as logically separate entities that happen to be configured in a particular way. In order for the overall system computation to be correct one would have to consider the composition of the behaviors of the components together with the interactions that are indicated by the lines. In terms of needs for abstraction, the lines in the first description represent programming language relationships such as "calls" (in the case of the i/o library) or "shares data structure" (in the case of the configuration description). In terms of needs for checking, type checkers will make sure that the required i/o routines exist in the library and that those routines are called with appropriate parameters. For the second description, we are also interested in typing questions: do the interacting filters agree on the type of data passing over the pipe. But, more importantly, we are concerned that the protocols of interaction are respected. For example, we would like to be able to check that a filter either reads or writes to a pipe, but not both; the filters agree on conventions for signaling end of data; the reader does not expect any more data than the writer will provide.

4 The WRIGHT Model of Architectural Description

Unfortunately, except in very specialized circumstances, the kind of architectural description given above is usually informal, and programmers must ultimately rely on concrete descriptions of implementation relationships to find the "truth" of the system.[2]

In order to support more direct specification and analysis of architectural descriptions, we have developed the WRIGHT architectural specification language. WRIGHT specifications are based on the idea that interaction relationships between components of a software system should be directly specifiable as protocols that characterize the nature of the intended interaction.

System Capitalize
    Component Split
        port In *[input protocol]*
        port Left, Right *[output protocol]*
        comp spec *[Split specification]*
    Component Upper
        port In *[input protocol]*
        port Out *[output protocol]*
        comp spec *[Upper specification]*
    ...
    Connector Pipe
        [Pipe specification]
Instances
    split: Split; upper: Upper; lower: Lower; merge: Merge;
    p1,p2,p3,p4: Pipe
Attachments
    split.Left **as** p1.Writer;
    upper.In **as** p1.Reader;
    split.Right **as** p2.Writer;
    lower.In **as** p2.Reader;
    ...
end Capitalize.

Figure 3: *Capitalize* in WRIGHT

Figure 3 illustrates the use of the notation for describing the example system. As shown, a system description is divided into two parts. First, both component and connector classes are specified, indicating the interface and computation of components and the protocol that the connector class represents. Second, the configuration of the system is described using instances of these classes, and the architectural topology is defined as a list of attachments.

A WRIGHT specification describes a component interface as a collection of *ports*, or logical interaction points.[3] These ports factor the expectations and promises of the component into the points of interaction through which the component will interact with its environment. The component may optionally further specify how the interactions on its ports are combined into a computation.

An example component specification (of the filter *Split*) is shown in figure 4. Each port is defined in terms of a protocol written in a subset of CSP [7].[4] In this example the *In* port of *Split* repeatedly reads data until it encounters a *read-eof* event, at which point it terminates. Similarly, the *Left* and *Right* ports repeatedly write data until it chooses to *close*. The **comp spec** specifies that the pipe alternately passes its

[2]For the example above we could have used a Unix shell language to directly describe the architecture of the system. But for other architectural designs, and even for more complicated pipe/filter designs that would not be possible.

[3]By using the term "port" we do not mean to imply that a port must be implemented as a port of a task in an operating system.

[4]See appendix A.

input to the *Left* and *Right* ports until a *close* event is encountered on its *In* port.

Component Split =
  port In = read?x→In [] read-eof→close→ $\surd$
  port Left, Right = write!x→Out ⊓ close→ $\surd$
  comp spec =
    let Close = In.close→Left.close→Right.close→ $\surd$
    in Close []
      In.read?x→Left.write!x → (Close [] In.read?x→Right.write!x → **computation**)

Figure 4: The Filter *Split*

A connector class is defined by a set of *roles* and a *glue* specification. The roles describe the expected local behavior of each of the interacting parties. The glue specification describes how the activities of the roles are coordinated. The use of roles and glue allows us to separate the requirements on each component (filling a role in the interaction) from the details of how the roles interact with each other. As we will see in the next section, this allows us to use localized port-role compatibility checking when we attach an instance of the connector in a system description.

Consider the specification of *Pipe*, illustrated in figure 5. This connector has two roles: *Writer* and *Reader*. The specification of the role *Writer* indicates that any participant at the source end of the pipe has two options: to write or to close the pipe, signaling the end of the communication. Because the writer is independent of the reader, the writer does not know or care that the data is delivered reliably to the reader using a FIFO discipline (as is indicated by the **spec** of the connector). Similarly, the *Reader* role indicates that this participant in the communication may decide to terminate the communication before all of the data has been delivered. The reader, however, must also account for the possibility that there may not be any more data to receive. This is indicated by the *Reader* role offering the connector a choice between the events *read* and *read-eof*.

The *glue* specification of a connector is divided into two parts: a finite-state protocol and a predicate over the traces generated by this protocol. In combination the two parts describe the nature of the interaction by constraining how the events in the roles are interleaved. The division into a "finite" part and an "infinite" part is analogous to the separation between a procedure's signature and its specification. As we indicate in the next section, the finite part allows us to check whether the connector is well formed, while the infinite part allows us to define its full behavior. For example, the *Pipe* specification in figure 5 indicates that data is delivered in FIFO order and that all of the data will be delivered before *read-eof* is signaled to the *Reader*.

By comparing the protocols of the ports, such as those of *Split*, with the roles of *Pipe*, we can observe a number of important points about our specifications. The ports *Left* and *Right* both indicate that *Split* reserves the decision about how much data to produce at that port (via the non-deterministic choice ⊓). Also, *Split* promises to consume all of the data presented to it at the port *In*. Note how this differs from the specification of the role *Reader*, which will be instantiated by this port in the configuration *Capitalize*. The *Pipe* connector permits a component the freedom to decide how much of the data to consume, but the *Split* component chooses not to take advantage of this freedom.

connector Pipe =
  role Writer = write!x→Writer ⊓ close→ √
  role Reader =
    let ExitOnly = close→ √
    in let DoRead = (read?x→Reader [] read-eof→ExitOnly)
    in DoRead ⊓ ExitOnly
  glue = **let** ReadOnly = Reader.read!y→ReadOnly [] Reader.read-eof→Reader.close→ √
                [] Reader.close→ √
      in let WriteOnly = Writer.write?x→WriteOnly [] Writer.close→ √
      in Writer.write?x→**glue**
        [] Reader.read!y→**glue**
        [] Writer.close→ReadOnly
        [] Reader.close→WriteOnly
  spec ∀ Reader.read$_i$!y • ∃ Writer.write$_j$?x • $i = j \wedge x = y$
    ∧ Reader.read-eof ⇒ (Writer.close ∧ # Reader.read = # Writer.write)

Figure 5: A Pipe Connector

5 Reasoning About Architectural Descriptions

An important property of any system description language is its ability to support reasoning about system descriptions. Standard MILs typically support reasoning about certain forms of consistency (such as typechecking) and (possibly) correctness.

In the case of architectural connection we are interested in richer forms of consistency. Specifically, we would like to be able to know when it is legal to attach a given connector as a port in some system description. We refer to this as the "port-role compatibility" problem. (Recall that we use an instance of a connector by associating its roles with the ports of instances of components — see figure 3.)

The most obvious and constrained form of compatibility checking would be to simply check that the port and role have identical protocols. But this is too restrictive. For example, we saw above that the ports of *Split* did not take advantage of the full flexibility provided by the *Pipe* roles. Similarly, we would like to be able to connect either end of a pipe to a file (as is done in Unix), even though files support both reads and writes while the end of a pipe supports only one. As another example, it should be possible to use a server port in a role that requires fewer services than the component provides. On the other hand we would not like the client port to attempt to use more services than the server provides.

Informally, we would like to guarantee that the promised behavior of a role is respected by the port. In CSP terms, when a process provides a deterministic choice to its environment the role should also do so. It turns out that this requirement is nicely captured by the notion of refinement in CSP (using the failures/divergences model), and with some straightforward modifications refinement can used in a definition of compatibility between roles and ports [2]. Further, it is possible to show that this notion of compatibility is sound in the following sense: if a connector is deadlock free then any compatible instantiation of an instance of that connector will remain deadlock free. An important practical consequence of this is that we can formally analyze a connector definition independent of its use and then perform simple local compatibility checks on each port-role attachment to determine if it is being used appropriately [2].

However, a formal definition of compatibility is not, by itself, enough. We would also like to be able to perform certain kinds of automatic checking of the property. To do this we need to have a sufficiently constrained description of the protocols. It is for this reason that we separated our connector role descriptions into two parts: a protocol (expressed as a CSP process) and a specification over the traces generated by that protocol. In a WRIGHT description the former is characterized with a "finite" subset of CSP.[5] The latter allows more general properties to be stated.

The result of these choices is that it is possible to use existing commercial tools to perform automated compatibility checking on WRIGHT descriptions. In particular, we use the FDR System [4], a tool that adapts model checking techniques to determine various properties of CSP processes (e.g., deadlock freedom and refinement assertions). Details on our use of FDR can be found in [1].

6 Related Work

Module Interface Languages

As we argued earlier, module interface languages, with their roots in programming languages, deal primarily with definition/use relationships. As such, the issues that they address are largely orthogonal to issues that Wright addresses. Indeed, to describe an *implementation* of one of the components in a Wright description, one would likely use a modern module interconnection language to define the code units and their dependencies.

On the other hand, because until recently software developers have had *only* traditional module notations to describe their systems, they have used the import/export facilities of these languages to define architectural structure as well. The result is that such "architectural" descriptions are necessarily limited to the interaction primitives of the programming language: usually procedure call and data sharing. In this respect, our work provides a significant improvement in expressiveness and analytic capability. It improves expressiveness because it allows the designer to specify new kinds of interactions that are not bound by the limitations of the programming model. Moreover, the description of these interactions includes specification of dynamic behavior (i.e., protocols). The inclusion of dynamic properties allows more properties to be checked than, say, signature matching alone.

Description of Software Architecture

A number of other research efforts are concerned with the problem of architectural representation. Luckham and Mitchell's Rapide language [9] uses the theory of Posets as its basis for architectural description. Like our work, Rapide has facilities for general description of communication patterns based on predicates over traces of events. Connector rules can be described using these predicates and connectors can be modelled as special kinds of components. Rapide supports certain kinds of static checking as well as run time checking of conformance to constraints. On the other hand, Rapide lacks a specific theory of connectors (independent of what it provides for components) and does not lead to automatable compatibility checking of protocols, as does WRIGHT.

Another increasingly popular approach is the use of domain-specific architectural description languages [11, 14]. These languages increase analytic and expressive leverage of system description by specializing the notation for a specific family of systems. The increasing descriptive power of these systems is encouraging, but the limited applicability to a particular domain make them unsuitable as a general model.

Finally, there are numerous software design languages that have been proposed — many of them graphical. These are concerned with the gross decomposition of systems (for example, [3]). As with domain-specific approaches to software architecture, such notations can provide strong support for certain classes of systems. But they do not provide general mechanisms for formal description or analysis of architectural connection.

[5] Essentially the expressive power is equivalent to finite state machines.

Protocols

Our work has taken the approach of viewing connector classes as describing communication protocols between components. There is a considerable body of work related to specification and reasoning about protocols [7, 8, 10]. Traditionally, research on protocols has been primarily concerned with developing algorithms to achieve certain communication properities – such as reliable communication over a faulty link. Having developed such an algorithm the protocol designer assumes that the participants will precisely follow the algorithm specified by the protocol.

Our use of protocols differs from this traditional approach to protocols in two significant ways. First, the protocols that define our connectors specify a set of obligations, rather than a specific algorithm that must be followed by the participants. This allows us to admit situations in which the actual users of the protocol (i.e., the ports) can have quite different behavior than that specified by the connector class (via its roles). This approach allows us to adopt a building-block approach, in which connectors are reused, the context of reuse determining the actual behavior that occurs. It also forces us to consider the problem of port-role compatibility, something that does not arise when talking about conventional protocols.

The second major difference is that our approach provides a specific way of structuring the description of connector protocols – namely, the separation between roles and glue. The benefits of adopting this structured approach is that it allows us to localize the checking of compatibility when we use a connector in a particular context. (Recall that local compatibility of the ports with their corresponding roles guarantees the preservation of deadlock freedom for the connector as a whole.)

One example of a use of protocols similar to ours is work by Nierstrasz on extending object-oriented notations to permit specification of object types in terms of protocols over the services that they provide [12]. Nierstrasz extends the definition of an object class to include a finite-state process over the methods of the object, and defines a subtyping relation and instantiation rules that are similar to our ideas of compatibility. While the motivation (and, in part, the techniques) are quite similar, our work differs in two ways. First, Nierstrasz considers only one kind of component interaction: method invocation. The need for a description of the glue therefore does not arise in his work. Second, the refinement relations used to define both subtyping and instantiation differ from the tests that we apply because they are specific to a single class of interaction. By focusing on this single kind of interaction, Nierstrasz can exploit the specific domain of object-oriented invocation to develop somewhat stronger theory, and in particular a subtype hierarchy. But his techniques are not as general as ours.

7 Conclusion

We have argued that the facilities provided by current MILs and IDLs for definition/use resolution are insufficient to capture architectural designs. What is needed is a way to express more directly the intended interactions between the high-level computational components of a system. Towards that end we have briefly outlined the WRIGHT specification language. WRIGHT allows one to define reusable connector classes that can be instantiated as needed to produce system descriptions. The meaning of a connector is given by a set of protocols, which state what are the roles of interaction and how these roles interact with each other. Further, we have indicated that this leads to a rich notion of compatibility (analogous to type consistency for current MILs) that is both expressive and checkable.

References

[1] Allen, R. and Garlan, D. *Formal Connectors*. no. CMU-CS-192, Carnegie Mellon Univ., 1993.

[2] Allen, R. and Garlan, D. *Formalizing Architectural Connection*. 1993.

[3] Cameron, J. **JSP and JSD: the Jackson Approach to Software Development**. IEEE Computer Society Press, 1989.

[4] **Failures Divergence Refinement: User Manual and Tutorial**. 1.2β. Formal Systems (Europe) Ltd., Oxford, England, 1992.

[5] Garlan, D. and Shaw, M. *An Introduction to Software Architecture*. **Advances in Software Engineering and Knowledge Engineering**, vol. I (1993).

[6] Guttag, J. V. and Horning, J. J. **Larch: Languages and Tools for Formal Specification**. Springer-Verlag Texts and Monographs in Computer Science.

[7] Hoare, C. **Communicating Sequential Processes**. **Prentice-Hall International Series in Computer Science**, Prentice Hall International, 1985.

[8] Holzmann, G. J. **Design and Validation of Computer Protocols**. Prentice Hall, 1991.

[9] Luckham, D. C., Vera, J., Bryan, D., Augustin, L., and Belz, F. *Partial Orderings of Event Sets and Their Application to Prototyping Concurrent Timed Systems*. March 1992.

[10] Lynch, N. A. and Tuttle, M. R. *An Introduction to Input/Output Automata*. no. MIT/LCS/TM-373, MIT Laboratory for Computer Science, November 1988.

[11] Metalla, E. and Graham, M. H. *The Domain-Specific Software Architecture Program*. no. CMU/SEI-92-SR-9, Carnegie Mellon Software Engineering Institute, June 1992.

[12] Nierstrasz, O. *Regular Types for Active Objects*. in: **OOPSLA '93**. **ACM Sigplan Notices**, vol. 28, 1993, pp. 1–15.

[13] Prieto-Diaz, R. and Neighbors, J. M. *Module Interconnection Languages*. **The Journal of Systems and Software**, vol. 6 (1986), pp. 307–334.

[14] **Proceedings of the Workshop on Domain-Specific Software Architectures**. Software Engineering Institute, Hidden Vallen, PA, 1990.

A A Brief Explanation of our Use of CSP

While CSP has a rich set of concepts for describing communicating entities, we use only a small subset of these, including:

- **Processes and Events:** A process describes an entity that can engage in communication events. Events may be primitive or they can have associated data (as in e?x and e!x, representing input and output of data, respectively). The simplest process, STOP, is one that engages in no events. The event $\surd$ is used represent the "success" event. The set of events that a process, P, understands is termed the "alphabet of P", or αP.
- **Prefixing:** A process that engages in event e and then becomes process P is denoted $e \rightarrow P$.
- **Alternative:** (also called "deterministic choice") A process that can either behave like P or Q, where the choice is made by the environment, is denoted $P \,[]\, Q$. (Here "environment" refers to the other processes that interact with the process.)
- **Decision:** (also called "non-deterministic choice") A process that can either behave like P or Q, where the choice is made (non-deterministically) by the process itself, is denoted $P \sqcap Q$.
- **Named Processes:** Process names can be associated with a (possibly recursive) process expression. Unlike CSP, however, we restrict the syntax so that only a finite number of process names can be introduced. We do not permit, for example, names of the form $Name_i$, where i can range over the positive numbers.

In process expressions $\rightarrow$ associates to the right and binds tighter than either $[]$ or $\sqcap$.
So $e \rightarrow f \rightarrow P \,[]\, g \rightarrow Q$ is equivalent to $(e \rightarrow (f \rightarrow P)) \,[]\, (g \rightarrow Q)$.

In addition to this standard notation from CSP we introduce three notational conventions. First, we use the symbol $\surd$ to represent a successfully terminating process. This is the process that engages in the success event, $\surd$, and then stops. (In CSP, this process is called SKIP.) Formally, $\surd \stackrel{\text{def}}{=} \surd \rightarrow \text{STOP}$.

Second, we allow the introduction of scoped names, as follows:

P = **let** $Q = expr1$ **in** R

Third, as in CSP, we allow events and processes to be labeled. The event e labeled with l is denoted $l.e$. The operator ":" allows us to label all of the events in a process, so that $l : P$ is the same process as P but with each of its events labeled. For our purposes we use the variant of this operator that does not label $\surd$. We use the symbol Σ to represent the set of all unlabeled events.

This subset of CSP allows one to define processes that are essentially finite state. It provides sequencing, alternation, and repetition, together with deterministic and non-deterministic event transitions.