Refining Interactions in a Distributed System
(Extended Abstract)

Neelam Soundarajan

Computer and Information Science
The Ohio State University
Columbus, OH 43210
USA

e-mail: neelam@cis.ohio-state.edu

Abstract:

We identify a new type of refinement that we call *interaction refinement* that is of great value in designing distributed systems. An interaction refinement step is one in which a system designer decides to implement a given 'high-level' interaction as a particular sequence of 'low-level' interactions. For example a particular high-level interaction may be specified as '*identify-client*' involving a 'client process' and a 'server process'. The designer may decide to implement this interaction as a series of low-level interactions between the two processes, with the client process first sending to the server process an *id*, followed by a *password*, and perhaps allow the client several attempts at getting the password right, etc. Here the designer is *not* refining the internal structure of either the client process or the server process; rather she is deciding to implement the 'identify-client' interaction between the two processes in terms of a sequence of low-level communications between them. By contrast, other types of refinment that are normally considered are concerned with refining the internal structure of one or more of the processes, the interactions between the processes being taken as a given.

We motivate the idea of interaction refinement with simple and natural examples, present a precise definition of the concept, and develop a preliminary formalism that can be used to establish correctness of systems developed using interaction refinement. We also compare our work with previous work which allow one to deal with restricted kinds of interaction refinement.

1 Introduction and Motivation

Stepwise refinement is considered, with good reason, one of the most important tools in software design. We believe there are three distinct types of refinement *Sequentiality refinement*, *Concurrency refinement*, and *Interaction refinement*. Sequentiality refinement, which we will abbreviate to *SR*, is the classic type of refinement introduced by Dijkstra, Wirth, and others, many years ago. In using *SR*, we refine the task of achieving a given goal starting from a specified initial condition, into subtasks for achieving subgoals, such that the given initial condition implies the initial condition of the first subtask, the subgoal achieved by the first subtask implies the initial condition required by the second subtask, etc., and the subgoal of the last subtask implies the original required goal. The process of refinement is repeated with each of the individual subtasks until the final subtasks can all be directly implemented using the primitives of the given programming language. Thus in using *SR*, we break down the original goal into a sequence of subgoals which together allow us to achieve the original goal; and we repeat this process with the individual subtasks. *SR* is a powerful tool and is used extensively, especially in designing sequential programs. Further, formal methods developed by Hoare, Dijkstra, Gries, and others, allow us to reason precisely about such programs, and the reasoning closely models the refinement steps used in developing the program. Indeed several authors have argued that the correctness proofs should be developed in parallel with the refinement steps of *SR*.

Concurrency refinement, which we will abbreviate to *CR*, is used, as its name indicates, for refining the concurrency present in a process. Thus, given that a process P is required to exhibit a certain set of behaviors perhaps in terms of the communications it engages in with processes external to P, we may be able to refine P into a parallel composition $[P_1 // \ldots // P_n]$ of processes $P_1, \ldots, P_n$, each of which is in turn required to exhibit certain behaviors, in the form of communications with each other and with processes outside P. And we repeat the task of refining the individual processes, until they can be easily implemented in whatever (concurrent) programming notation we are using. In fact, *CR* and *SR* are usually combined; thus we may decide to design, say, P_2 as a sequential process. *CR* is much harder to use than *SR* which is just another way of saying concurrent programs are much harder to design than sequential programs. Not surprisingly, the formalisms for establishing that these programs do exhibit the behavior we desire, such as those of [7, 1, 6, 8] are much more involved than the earlier formalisms for dealing with programs developed using only *SR*.

The third kind of refinement, and the focus of this paper, is *Interaction refinement* or *IR*. We believe that *IR* plays a critical role in the design of distributed systems. Indeed, as the examples in the paper will show, designers use it all the time although, to our knowledge, it has not been precisely defined or even explicitly identified, nor have any formalisms been developed to deal with the refinements that developers come up with using *IR*. The goal of the paper is to informally explain what *IR* is and why it is so important in the design of distributed systems, provide a formal definition for it, and propose a proof rule that can be used to establish the correctness of systems developed using it.

Consider how a distributed system for a particular task might be designed. To start with, the designer typically has a good idea of the key processes in the system, as well as some high level idea about what kinds of interactions ought to take place between these processes. Consider, for instance, a system in which a 'client process' *cp* will interact with a 'server process' *sp*, after first identifying itself to the latter. At a high level the system designer might express this by specifying that the first interaction between *cp* and *sp* serves to verify the identity of the client. As the system design proceeds, the designer may decide to map this identification task into a series of interactions between the server and client, and may even decide to involve other processes in this task. For instance, *cp* may be required to provide a password which the server then authenticates, possibly using an 'authentication process' *ap* that the designer decides to include in the system. Depending on the details of the design, the authentication will require interactions between *cp* and *ap*, or between *sp* and *ap*, or possibly all three. These interactions were *not* part of the original specification. Rather each set of interactions is a different *refinement* of the original specification and corresponds to a particular design (other designs will correspond to other refinements of the original specification). It is this type of refinement that we call *interaction refinement* or *IR*.

Note that this type of refinement is quite different from the situation where say the process *sp* is built as a network of cooperating processes that are *internal* to *sp* (which would then be an example of *CR*). Almost by definition, these processes, as well as any communications between them, would be invisible to external observers – including processes like *cp*. If we were to redesign these internal processes, that would have no effect on *cp* so long as there are no changes in the external communications of *sp*. By contrast, in the situation we are considering the designer is refining a 'high level interaction' between certain processes in the system into a series of 'lower level interactions' between these and other processes. It is true that in the current example we could encapsulate the authentication process *ap* inside *sp*. In that case, *cp* would only be aware of, and interact with, *sp*. But we would still have to deal with the refinement of the high level interaction 'identify-client' into a series of lower level interactions involving such actions as *cp* sending a password to *sp*, and *sp* possibly allowing *cp* several attempts to get the password right, etc.[1] Moreover, including *ap* as an internal process of *sp* is really a different design; in particular it lacks the flexibility afforded by the other design of changing the authentication algorithm by simply replacing *ap* (with a process that uses a different set of algorithms). Hence the designer must be free to choose either of these designs, and our formalisms must be able to deal with both.

The key question is, how do we know that a particular interaction refinement is correct? We will address this question in the next section after first introducing the trace based notation that we will use for specifying interactions, and defining precisely the notion of interaction refinement. We then present a proof rule that will allow us to formally show the correctness of a given refinement. In the third section we present (informally) a couple of examples that we believe show how widely interaction refinement is used in the designing distributed systems, and how the formalism of section 2 can be used to precisely show the correctness of particular refinements. In the final section we reiterate how interaction refinement differs fundamentally from other kinds of refinement while at the same time –as will be clear from the formalism of section 2– it is a type of abstraction/concretization just like the other kinds of refinement. We try to argue that interaction refinement will provide an important new approach to designing and establishing the correctness

[1] On the other hand, when using *CR*, we do not try to 'refine' the external communication behavior that a process is expected to exhibit; that is a given. Rather we try to refine its internal structure, so that it will exhibit the specified external communication behavior.

of distributed systems. We also compare our work with earlier work [4, 5] that allows us to deal with certain restricted kinds of interaction refinements.

2 Specifying and Verifying Interaction Refinements

We will use a *trace-based* approach to specify processes. With each process P we will associate two trace variables: ε will record all external interactions that P engages in; δ will record both external and internal interactions of P. If P has no internal processes, these two traces will of course be identical to each other. In general, ε can be obtained by projecting out of δ all elements that correspond to internal interactions of P. We will occasionally use τ to stand for either kind of trace, and let the context indicate whether we are considering the internal or external trace. (If we are considering the trace of more than one process, we will subscript the τ with the identity of the process to distinguish between them.)

An interaction is often, but not always, a communication in which one process sends a specific value to another. In general, low level interactions tend to be communications, while higher level interactions tend not to be. Thus in the example of the client process *cp* and server process *sp* of the last section, 'identify-client' is a high level interaction but not a communication, whereas the low level interaction in which *cp* sends a password to *sp* (or to *ap*) is a communication.[2] We will use an element of the form (P, Q, v) in a trace to record a communication in which the value v was sent by the process P to the process Q. We will record an interaction that is not a communication as an element that specifies the name of the interaction, the identities of the processes involved, and any parameter values that may be needed. Note that a communication involves exactly two processes, whereas a general interaction may involve two or more processes.

The specification of a process will be an *invariant* on the trace of the process.[3] Consider a system S consisting of a number of processes. A high-level specification of this system would be an invariant I on δ imposing appropriate restrictions on what interactions may take place between the component processes of S, as well as the interactions between S and external processes, and the relative orders of these interactions. Suppose we have another specification I' for S. We can now precisely state the question from section 1: What relation must hold between the specifications I and I' in order for us to be able to legitimately claim that I' is a (interaction) refinement of I?

Note first that the obvious answer, that I' is a refinement of I only if it implies I, is wrong or rather not general enough. Given our previous discussion, the elements in the trace that I is concerned with are not, in general, the same kind of elements as the ones in the trace that I' is concerned with. Rather the former correspond to what we have called high-level interactions, whereas the latter are low-level interactions (possibly communications) that *implement* those high-level interactions. So even if I' were a proper refinement of I, an implication relation will in general

[2]This may be debatable; it may be that sending a password is actually quite a complex interaction involving a number of communications between the client and the server. This just goes to show that interaction refinement, like other types of refinement, is in general a multi-step activity.

[3]For simplicity we are considering only *safety* properties, thus invariants are sufficient. The main ideas in this paper will however apply to other kinds of properties than safety, and to other kinds of specifications than invariants.

not hold.[4] In view of this difference between the elements in the traces, let us use δ to denote the trace that I refers to and δ' to refer to the trace that I' refers to. Note again that both δ and δ' are (internal) traces of S and record all interactions between the various component processes of S as well as interactions with processes external to S. The difference is that the view recorded in δ is a high-level view whereas that in δ' is a low(er)-level view.

Let us first consider a simple kind of relation between the elements in δ and δ'. Let Σ be the set of elements corresponding to the high-level interactions, i.e., that can appear in δ, and Σ' the set of elements corresponding to the low-level interactions. A simple situation would be for the interaction represented by any given element of Σ to be implemented by a sequence of elements of Σ'. Thus we will need a mapping of the form:

$$\rho : \Sigma \Rightarrow \Sigma'^*$$

where Σ'^* is the set of all (finite) sequences over Σ'.

If $\rho(i) = \langle i_1, \ldots, i_n \rangle$, that means that the high level interaction i is being implemented as the sequence of low-level interactions $i_1, \ldots, i_n$. Note that there is no assumption that only the processes involved in the interaction i are allowed to be participants in interactions $i_1, \ldots, i_n$. Indeed, it is not even required that all (or any!) of the participants in i be involved in one or more of $i_1, \ldots, i_n$. This may seem rather strange; while it may be appropriate to allow additional processes to participate in the implementation of an interaction between a given set of processes, how can an interaction that at an abstract level involves certain processes be 'implemented' at a lower level without the participation of those processes? The answer is that this is a matter of system design and the formalism should not disallow such possibilities. For instance, it may be that the nature of the system is such that this interaction is always preceded by certain other interactions involving certain other processes and hence these other processes can consult with each other to implement the required interaction without any apparent involvement of certain of the principals.

Once the mapping function is given, we can see what relation must hold between I and I'. Essentially we need to ensure that if a given trace δ' satisfies I', then the corresponding δ trace will satisfy I:

$$\forall \delta'.\{I' \Rightarrow \{\exists \delta.(\delta' = \rho(\delta) \land I)\} \tag{1}$$

where $\rho(\delta)$ is the sequence obtained by applying ρ to each element of δ and then appending together all the resulting sequences. Thus it is the natural extension of ρ which is defined over elements that correspond to 'high-level interactions' to sequences of such elements.

Rule (1) is a natural generalization of the usual implication relation between a specification I and its refinement I' to take account of the fact that during interaction refinement high-level interactions are implemented in terms of lower level ones. In particular if there is no difference between the high-level and low-level interactions, the function ρ reduces to the identity map, and (1) reduces to the usual implication relation.

The mapping function ρ and the implication (1) are however a bit too restrictive. They require

[4]Of course, if I' does imply I, then I' is indeed a proper refinement of I but this is the simple kind of refinement that has been considered so far in the literature; we are trying to generalize this to the notion of interaction refinement. We will see shortly that this simple kind of refinement is indeed a special case of our definition.

that *every time* a given high-level interaction is implemented, it must be implemented in exactly the same way as the previous time. This would prevent refinements that exploit past history in implementing particular interactions. Suppose, for instance, that a system has two processes P_1 and P_2 and that the high level requirement says that P_1 must send a series of documents to P_2. If in practice it turns out, as it does in many applications, that frequently a document that P_1 sends to P_2 is a slight revision of (or even identical to) the previous document that it sent, most designers would implement this by having P_1 send only this differential information. The high-level specification would state that P_1 sends a series of documents, but the lower-level implementation achieves the effect much more efficiently. This is exactly the sort of thing we want to be able to deal with but the formalism we have developed so far will not allow for it, since according to the ρ function, a given high-level interaction must be mapped to a fixed set of low-level interactions.

The solution is clear from the discussion above. The function ρ should not be a function over the *individual* high-level interactions but rather over the high-level traces:

$\rho : \Delta \Rightarrow \Delta'$

where Δ is the set of all possible high-level traces (essentially the set Σ^*), and Δ' the set of all possible low-level traces. Requirement (1) doesn't need any change except to note that the function ρ that it refers to is this new one that maps δ traces directly to δ' traces.

Naturally we will have to impose some conditions on the function ρ. For instance, a reasonable requirement on ρ would seem to be monotonicity, i.e., if δ is a prefix of δ_1, then $\rho(\delta)$ must also be a prefix of $\rho(\delta_1)$, but we won't go into these complexities here except to note that we have to be careful not to impose restrictions that disallow refinements that designers find useful. Thus one possibility we need to explore is whether a given high-level trace δ must necessarily map to a unique low-level trace δ' as implied by the ρ above, or would more general relations be better? Or again, if i and i' are are two consecutive high-level interactions in δ, isn't it conceivable that in some reasonable refinement that the corresponding elements in δ' are interleaved with each other rather than all the elements corresponding to i appearing before any of the elements corresponding to i'? And if that is true, wouldn't this violate the monotonicity requirement? These are questions that we hope to explore in a future paper.

3 Some Examples

We believe that *IR* is used extensively in the design of distributed systems. We consider two examples in this section, and show the application of *IR* in the design of each. Our discussion of these examples is quite informal and far from complete, the main point of the section being to show the role *IR* plays in the design of these systems.

For our first example consider a pair of *fax machines* M_1 and M_2 connected by a telephone network T.[5] Suppose M_1 wishes to send a document to M_2. At a high level, this would require a sequence of three actions: Establish a connection between M_1 and M_2, send the document from M_1

[5]This example is inspired by Buhr and Casselman's book [2] on '*use-cases*' for dealing with object-oriented systems. Use-cases, since they record the sequence of method calls to various objects in an OO system, are similar to

to M_2, and finally break the connection between M_1 and M_2. How do we refine these interactions at the next level? One thing we definitely need to do is to introduce the telephone network T into the picture (since faxing documents using smoke signals is not yet a well developed technology! and if it were, we would have to involve the atmosphere, the smoke signal network, etc. into the picture). Note that although T was not mentioned in the high level interactions, nor is it an internal component of either M_1 or M_2, it is entering the picture in the low-level refinement; as we said before, this is typical of ***IR***. Depending on the details of the telephone system each operation can now be broken down into a sequence of actions. Thus establishing the connection may involve the sending of the area code from M_1 to T, followed by the number for M_2; T will then send a message to M_2 which acknowledges and accepts the connection (picks up the phone), T then sends a message to M_1 indicating the the connection has been set up. Note that the details of how T maps the given area code and the number to the process M_2 are *internal* to T. Conceivably, T might be internally organized as a collection of cooperating processes that interact with each other for this task. This refinement, being internal to T, is *not* an example of ***IR***; rather it is an instance of ***CR***.

Next we can define the high-level action of M_1 sending the document to M_2, or of tearing down the connection between M_1 and M_2. We will omit the details but we should note that other designs are possible. For instance, a smart T might decide to simply accept the document sent by M_1 and store it internally, forwarding it to M_2 at a later time when traffic is low. This is a different refinement, and will be represented by a different ρ function in our formalism.

For our next example, let us briefly consider the classic layered network protocol. Here we have two processes P_1 and P_2 that need to communicate. At the highest level, the interaction is specified as a series of interactions in which P_1, say, sends values to P_2. As we go down the layers, more and more interactions between P_1 (or rather its internal components) and P_2 (its internal components) are introduced. Note that these layers *are* instances of ***IR*** since these communications are not internal to P_1 or P_2. Indeed, each layer is essentially a new level of interaction refinement. The interactions introduced at a given layer implement –rather than being identical to– the interactions at the next higher layer. In our formalism, this will be represented as the ρ function corresponding to this layer.

It is worth repeating here that when using *interaction refinement* what we are doing is to decide to implement a desired 'high-level interaction' (or a sequence of such interactions) by a particular sequence of 'low-level interactions'. We are *not* designing the internal *structure* of any process, as we would be if we were using one of the other refinement procedures. Thus ***IR*** and the other two kinds of refinement are really complementary and are all needed when designing a complete system.

traces. The main point of [2] is to develop pictorial use-case-maps that can be used in an informal manner to understand the behavior of OO systems. Buhr and Casselman do note that a *set* of points –each of which corresponds to a method call– rather than each point on a use-case-map, might represent a meaningful operation. They do not, however, address the question of how this operation might be identified or how we might show that this set of points does correspond to it, etc.

4 Discussion

We have identified *interaction refinement* as an important tool in the design of distributed systems. And we have provided a preliminary formalism that can be used to establish the correctness of systems designed using *IR*. As noted before, although the explicit identification of *IR* seems to be new, designers have been intuitively doing this type of refinement for a long time. But in the absence of a formalism like we have proposed here, it has been rather difficult to show correctness of the resulting systems. This should not be surprising. After all if a specification of a system requires the process P_1 to send a series of documents to P_2, and the designer implements this by having P_1 send only the differential information between consecutive documents, how can we consider this a correct implementation? We cannot, unless we introduce the idea that a specified interaction or sequence of interactions can be 'refined' into a different sequence of interactions, and identify precisely the conditions that must be satisfied in order for such a refinement to be considered correct. That is what we have tried to do.

Interestingly, there have been some attempts to establish the correctness of systems that have clearly used interaction refinement without depending on a formalism that allows for this. For instance, Creveuil and Roman [4] tackle the problem of verifying the correctness of a message router. They go through a series of design steps to arrive at their final router, and several of these steps are interaction refinements; in particular these steps introduce additional communications in the system. Creveuil and Roman formally establish using the approach of [3], that their router meets its specifications. They are able to do this because their router's high-level specification has been carefully designed in such a way that it *allows* additional communications –such as the ones introduced in their design– since it doesn't forbid them explicitly! Thus formally one can deal with this specification as if we are talking about a system that includes these types of communications, but that the specification imposes no special conditions on them! Clearly such an approach will not work if a particular interaction refinement in any way altered any of the high-level interactions, as in the case of the example considered in the last paragraph.

Before concluding, we should mention also the work of Francez and Forman [5] on *Interacting Processes*. They generalize the notion of communication between two processes to a multi-party interaction. Their language notation allows us to define a *team* which is a kind of template for a particular kind of multi-party interaction. Then whenever a set of processes wishes to engage in this type of interaction, they all 'enroll' in the particular team. Is such a team the same as an interaction refinement? In other words, could we consider the multi-party interaction as a high-level interaction, refined by the low level details specified in the corresponding team? Although there seems to be some relation, we believe a team is not an interaction refinement. Rather, a team is a particular implementation of the various communications needed to achieve the multi-party interaction. It is certainly a refinement, but it seems to be more an instance of concurrency refinement, rather than interaction refinement.

5 References

1. K Apt, N Francez, W deRoever, A proof system for CSP, ACM TOPLAS, 1980.
2. R Buhr, R Casselman, Use case maps for OO systems, Prentice-Hall, 1995.
3. KM Chandy, J Misra, Parallel program design, Addison-Wesley.
4. C Creveuil, GC Roman, Formal specification and design of a message router, ACM TOSEM, vol. 3, pp. 271-308, 1994.
5. N Francez, I Forman, Interacting processes, Addison-Wesley.
6. J Misra, KM Chandy, Proofs of networks of processes, IEEE Trans. on Software Eng., 1981.
7. S Owicki, D Gries, Verifying properties of parallel processes, CACM, 1976.
8. N Soundarajan, Axiomatic semantics of communicating sequential processes, ACM TOPLAS, 1984.