

A case for refinement calculi in software architecture design.

C. Montangero

Dept. of Informatics - University of Pisa

Corso Italia 40

56125 Pisa, Italy

+39 50 877245

monta@di.unipi.it

ABSTRACT

We argue that the research agenda for software architectures should leave room for inquiries into architectural refinement calculi.

A first good general reason for this is to avoid old errors: architectural refinement calculi should help in getting the right architecture right from the outset, rather than rely only on afterwards checks.

A second reason, which is more specific, and directly addressed in this paper, is that formal refinements can be actually effective only if they come with simple, workable concepts and notations. Therefore, we want to show (i) that there is room for investigations in more abstract concepts than those already proposed to describe architectures, like connectors, and (ii) that the framework provided by a calculus can be of great help in assessing the adequacy of these same concepts.

We note that the statements above carefully avoid negating the value of double checking, that is the need for analysis and testing, even in the presence of a formal architectural derivation: we only want to foster the interest for formally documented architectural developments.

Keywords

Refinements, remote actions, interactions.

1 INTRODUCTION

This position has been written with software design in mind, that is we are interested in how software architectures can best influence the design of quality software. We are not addressing the use of architectures for reverse engineering, or other similar uses, even though one may maintain that these area also might benefit from the techniques and concepts developed with design in mind.

We also have in mind a specific design process, namely the one based on formal refinements, where the complete justification of the development a product, a software architecture in our case, is recorded in a document. This process should be contrasted with the standard one, where

the emphasis is on the outcome of the effort, i.e. the description of the software architecture, which may be accompanied (but often is not) by a rationale. Tools currently supporting this modus operandi include the B-tool [LH96], and to a lesser extent MathPad [MathPad]. The document resulting from the refinement process is usually such that the description of the development goal, e.g. the architectural description, can be automatically extracted.

The traditional benefit of the formal refinement approach is precisely this natural intertwining of development and documentation. One of the purposes of this paper is to single out some of the issues related to the extension of the approach, traditionally applied to programming in the small, to programming in the large.

The exercise will help at least us to clarify some issues related to software architectures. I hope that the blunt attitude, that we will sometimes take, will rise lively discussions at the workshop. The main point we are making is that current architecture description languages take a too low-level look at the connections among components, and that calculi may help in establishing the right level of abstraction.

2 THE ISSUES

The purpose of the architecture design lies in singling out the major components of a system and their interactions. Our position here is twofold:

- (i) the semantics of the components is paramount, in defining their interactions, and
- (ii) ports and roles (or similar concepts) are too detailed to introduce them from the outset, when architecting the system.

That is, we need a formalism that makes it easy to express the link between the semantics of the components and their interactions.

As an example, consider the great grandmother of all connectors, the pipe. The design choice of splitting a component in two, which interact via a pipe, is reasonable only if the goal of the original component can be split accordingly. That is, in such a way that a (finite) stream of data can be meaningfully passed from the first

new component to the second one. Hence, the essence of the design step is to identify the stream, not the ports through which the stream will flow.

So, we should be looking for a framework in which the architectural design step 'introduce two components interacting through a pipe' makes it easy to verify that the newly introduced stream is indeed sufficient to achieve the goal of the original component.

More in general, we are looking for a framework in which it is easy to check that the expected behavior of a component to be split in a development step can be implemented via the *stereotypical* interactions of the components introduced in the step.

For the program to succeed, much attention has to be put not only on the concepts, but also on the notations and their ease of use. Morgan [M94] is a good source of inspiration: in fact, the main proposal here, the *remote action*, can be seen as the natural extension of the *abstract programs* he introduces. Connectors are a way to characterize the interactions between components: the position taken here is that they tend to introduce too many details too early, therefore impeding a smooth refinement. In fact, what an interaction is about, as captured by the behavior of a connector relative to its roles, is affecting the partner remotely. Just as Morgan proposes to consider in the same framework specifications and code, by regarding specifications as abstract programs, we propose to abstract away from the details of the connections, by abstracting a connection to the explicit remote effect of an interaction on the partner.

3 AGAIN THE EXAMPLE

Consider the following component specification, in a WRIGHT-ish style, which is inspired by Moriconi's running example [MQR95]: the architecture of a compiler. It describes a component, whose goal is to produce an abstract syntax tree (AST), *t*, from a string of characters, *s*. Below, the names of the channels are motivated by what comes next, and should be read as pre-condition event and post-condition, respectively.

```
Component D
  Port In = prev?s -> In
  Port Out = post!t -> Out
  Comput = In.prev?s -> Out.post!t -> Computat
```

In Moriconi's example, the design decision is taken, to split D into a token generating component and a parsing component, which actually generates the AST. This refinement step might be represented in the WRIGHT-ish setting, by changing the specification of D somewhat like (assume that *mid* is the channel along which the newly parsed tokens are sent, in variable *i*, and *N* a suitable constant; the star is used to denote bounded iteration).

```
Component D
  Port In = prev?s -> In
  Port Out = mid!i -> Out
  Comput = In.prev?s -> [*j:1..N.Out.mid!i]
-> Computat
```

The point here is that this specification entails that no interaction between D and the parsing component occurs on the pipe fed by Out, until all the input string is available. Besides, any other solution would also bias the design of D, and anyway introduce too many design details too early. What is needed at this design stage, is the characterization of the stream that allows D to interact with the newly introduced parser, and nothing more! This is where the introduction of remote actions helps us.

4 REMOTE ACTIONS

The idea of a remote action is grounded in asynchronous communication and remote write, as you find them in the literature on multiple tuple-spaces. However, rather than taking a programming approach, and considering the implementation constraints to design an (efficiently) executable language, we allow to express the overall effect of communications and writings, as if it were the effect of a single remote action.

Though remote actions can be liberally used in principle, we claim that the best result can be obtained by restricting to well established patterns of use, i.e. to architectural styles [MS96b].

Morgan's abstract programs should be extended to include remote post-conditions (according to the evidence provided by the current state of our research: remote pre-conditional events might be considered as well...). Then, channels and ports are code, under reasonable assumptions. On the other side, there is no need to introduce such a level of details, unless they are really needed.

The exhibition of channels and ports at the architectural level manifest separate compilation issues at too high a level, just as goto's manifest machine level concerns at the algorithmic level.

An analogy may be useful here: when calculating structures to building bridges, hinges have a very simple characterization: a hinge is only able to resist to a force along the rod insisting on it. But, have a look at a real hinge, under a bridge in London, to understand that there is a lot of (detailed) design going into putting a hinge into operation. Analogously, there may be a lot of design into putting a pipe in place, like fixing ports, etc.

Obviously, the problem of reducing remote actions to code, that is to patterns of communications which can be

actually executed by interconnected machines need to be solved once per each supporting communication paradigm, and also only once per refinement pattern. This is the main advantage of considering remote actions in the framework of software architectures.

5 OUR APPROACH

There is some evidence of the usefulness of temporal logic in the description and implementation of parallel and distributed systems [CM88, RGB93, McCR97]. Also, Apt and Olderog suggest that the full power of temporal logic is needed, when in their monumental work on verification [AO91] they recognize the impossibility of analyzing liveness properties in their weaker framework.

Therefore, we assume a temporal logic with powerful operators extending those of Unity to deal with events. An event occurs when a given condition on the state of a component is established. The only operators we need in our examples are *causes* and *after*: *causes* relates an event and a condition, and specifies that the occurrence of the event is sufficient to arrive in a state in which the condition holds; *after* also relates an event and a condition, but specifies that the condition is a necessary one for the event to occur, namely that the condition must hold in the state before the event occurs.

A component has a name, and its behavior is described by liveness (something good will happen) and safety (nothing bad will happen) properties on their state. Formulae can be qualified by component names, and this feature is also used to express remote actions. Preliminary work in this direction is in [SS97, MS96, MS96b]. We are now ready to see the sketch of the pipe style at work, and have a look at the kind of refinement laws that should be available for architecture calculation.

This is the kind of refinement patterns we would like to see applied to introduce an order-preserving producer-consumer interaction:

```
D:[prevcauses T:post]

<<          con N and
P:(Ai:1(N-1). mid(i+1)after mid(i))

D:[prev causes P:(&i:1..N.mid(i))]
|: [R(&i:1N.mid(i)) causes T:post]
```

The annotation on the same line as the 'refines to' symbol <<, introduces logical constant N, and also states an invariant, that is a safety property. It states the characteristic property of the remote action: the consumer will receive its bits of input in the intended order. However, it does not constrain the producer to generate them in the same order: obviously, this might well be the

best strategy to follow to implement the interaction. But this is another story (to be told later in the refinement).

In the compiler example, the three specifications should be read:

- 1) the event 'string available' (*prev*) is sufficient to reach a state in which 'the abstract syntax tree is available to T' (*post*)
- 2) the event 'string available' is sufficient to reach a state in which 'all the tokens are available to D' (*mid(i)* being 'the i-th token is available')
- 3) the event 'all the tokens are available to P' is sufficient to reach a state in which 'the abstract syntax tree is available to T'

Here, we are not forcing D to wait until *prev* to start interacting with P: this specification is loose enough to allow, when designing D, to weaken *prev* and strengthen *P:(&i:1N.mid(i))* in order to achieve an efficient exploitation of the interaction.

The result of the refinement can also be described in a graph, like

```

D
<<
      mid( ), N{<}
D p-----c P
```

where the decorations on the tips specify the basic kind of interaction, and < reflects the safety invariant, which forces the order to be preserved.

The problem with abstract programs is that they may not be refined to code, that is they are miracles. The same likely happens with remote actions, and some feasibility condition should be devised.

6 A DETOUR ON REFINEMENTS

It is only when approaching code that the invariant introduced by the refinement pattern above has to be guaranteed: it will result in a constraint on the connector eventually introduced to implement the interaction. In the compiler example, the obvious way to weaken *prev* is to introduce an event *prevt(i)* as enough characters are available in D to produce the i-th token, such that

```
(&i:1..N.prevt(i)) => prev
```

and then refine D via a *pipelining law*, which should be, reasonably enough, something like

```
D:[ (&i:1..N. prevt(i)) causes
P:(&i:1..N.mid(i))]

<<          con I

D:[prevt(i) &I<N causes i=I+1 &P:mid(i)]
```

initi=1]

7 FINAL REMARKS

Although the plan outlined above seems sound enough, and there is evidence that it might be successful, its details have been worked out only to a limited extent, and no attempt has yet been done to extensively characterize the known patterns of interaction in this way. We argue that this endeavor should be undertaken, to provide the future software designers with better conceptual tools.

There is only one work we are aware of that deals with formal architecture refinements, namely [MQR95]. Besides not satisfying the criterion of simplicity, their approach is different from ours. Indeed, they are interested in deriving an architecture description from another, with operations that remind more of optimizing state transformations than refinements. The example they tackle is the transformation of the pipe implementing the interaction we discussed above into a symbol table. In our approach we would not transform the pipe into a symbol table, but rather start a new line of refinement after the introduction of the order preserving producer-consumer interaction.

The relevance of optimizing transformations at the architectural level is an open issue. In fact it links with the other big open issue, that is how to augment the quested calculus with annotations for non-functional properties, like performance bounds, time-liness properties, robustness, etc.

Acknowledgments

L. Semini provided insightful comments on an earlier draft.

REFERENCES

- [MQR95] M. Moriconi, X. Qian, and R.A. Riemenshneider, Correct Architecture Refinement, *IEEE Trans. on Software Engineering* 21, 4, 1995, 356-372.
- [M94] C. Morgan, *Programming from Specifications*. Prentice-Hall, Cambridge, 1994.
- [LH96] K. Lano and H. Haughton, *Specification in B, An introduction using the B Toolkit*, Imperial College Press, London, 1996.
- [AG96] R. Allen and D. Garland, A Case Study in Architectural Modeling: The AEGIS System, *Proc. 8th Intern. Workshop on Software Specification and Design, IWSSD8*, Schloss Velen, Germany, March 1996, 6-15. IEEE Computer Society Press.
- [CM88] K. M. Chandy and J. Misra, *Parallel Program Design: A Foundation*, Addison-Wesley, Reading Mass. 1988.
- [RGB93] G. C. Roman, R. F. Gamble, and W. E. Ball, Formal Derivation of Rule-Based Programs, *IEEE Trans. on Software Engineering*, 19, 3, March 93, 277-296.
- [McCR97] P.J. McCann and G.-C. Roman, Mobile UNITY Coordination Constructs Applied to Packet Forwarding for Mobile Hosts, *Proc. Coordination '97, 2nd Intern. Conference on Coordination Languages and Models*, LNCS 1282, Springer, 1997, 338-354.
- [AO91] K. Apt and E.-R. Olderog, *Verification of Sequential and Concurrent Programs*, Springer, 1997.
- [SM97] L. Semini and C. Montangero. A Refinement Calculus for Tuple Spaces. Available from ftp://ftp.di.unipi.it/pub/Papers/montangero/oikos_tl.ps
- [MS96] C. Montangero and L. Semini, Applying Refinement Calculi to Software Process Modelling. *Proc. 4th Int. Conf. on the Software Process ICSP4*, Brighton, UK, Dec. 1996, IEEE Computer Society Press, Los Alamitos, 63-74.
- [MS96b] C. Montangero and L. Semini, Refining by architectural style or Architecting by refinements. *Joint Proceedings of the SIGSOFT '96 Workshops, Part I, 2nd Int. Software Architecture Workshop*. S. Francisco, Oct. 1996, ACM Press, 76-79.
- [V98] R. Verhoeven, The Official Homepage of the MathSpad Editor, Available from <http://www.win.tue.nl/~wp/mathspad/>