AGENTS AND CONCURRENT OBJECTS

Jean-Pierre Briot Interviews Les Gasser
(To appear in IEEE Concurrency, 1998)
DRAFT 0.7 of 9/29/98

AgentsAndObjects-07.doc [Word6 Version]

Introduction:

Within the current special series on actors and agents, we conducted an interview with Les Gasser, a leading figure in multi-agent systems. Les Gasser is currently an Associate Professor in the Graduate School of Library and Information Science at University of Illinois at Urbana-Champaign (UIUC) and program director at the National Science Foundation (NSF). Our aim was to ask him about what he recognizes as the main contributions of the notion of agent and how it is different from (object-oriented) concurrent programming.

BRIOT: Let's start with the idea of agents - what do you think makes that a new or interesting idea?

GASSER: I think many people talk about agents without clearly specifying what may differentiate agents from other kinds or forms of programmed entity. So pretty often the discussion about agents focuses on things that don't seem to be a great deal different from distributed objects---for example, some independent existence in some way, distributed locus of control, encapsulation and autonomy, the ability to communicate and interact with other entities in a structured way, long-term organized relationships with other such objects, and so on. So I think a more direct way to see the prospects for a concept like 'agents' is to go back to the history of programming and locate the concept of agent in a historical progression of programming---a progression of techniques for description and action. And once we do that, maybe we can see what may be the contribution of agents, as a part of an evolution of programming languages and programming technologies.

BRIOT: So you'd prefer to take a historical viewpoint toward agents, rather than just defining them as something completely new?

GASSER: Right. We should look at agents as a natural extension of some streams in the history of programming, as well as communications and control too. Of course, there are also basic driving ideas from other intellectual areas in computer science, such as Artificial Intelligence (AI) and Distributed AI (DAI), human-computer interaction, Computer Supported Cooperative Work (CSCW) and so on. In fact one of the exciting things about agents R&D is that it's so synthetic of information from a wide collection of disciplines. You can also include organization theory, language theory, economics, psychology, and on and on. In a way, I think the excitement on the research front surrounding agents stems from this issue of integrating knowledge---it's intellectually interesting and it's a real challenge!

BRIOT: I agree. But for the moment let's just concentrate on the relationship between agents and programming technology. What are your ideas on this?

GASSER: The basic point is that, over time, programming technologies have moved ahead on two fronts. First, they've had to become more and more structured, with more and more conventions---there are now clearly right and wrong ways of designing programming tools, of and building programmed systems. This has been a way of managing the openness and complexity of modern computing environments and applications. The structure didn't come from nowhere---it's the product of years of trial and error and theory. Second, the particular structures and conventions that have evolved turn out to embed more and more knowledge about the context of programs, about the environments in which they live and work. All this structure and convention of a modern programming environment is only possible because programming technologies now incorporate a whole lot of knowledge about the environments in which programs will be used.

BRIOT: Can you get more precise?

GASSER: Suppose we go back to the 1950's, and think about machine languages and assembler code. These languages are extremely general and flexible, and they're linked directly to the capabilities and organization of specific machines. They're very concrete, and they're also extremely general. They give you maximum freedom of expression and maximum control over the machine. In fact they're still used for this! But machine languages are also very hard to understand and control. It's truly hard to program anything complex in a machine language, precisely because of all that freedom. So freedom comes at a price: you can't handle complexity, and you don't know how to map the language's possibilities to capture the structure that's already been built into the application domains you're trying to deal with. This is because machine languages embody a great deal of knowledge about the behavior of the machine, but almost no direct knowledge or structure that reflects the objectives and constraints of the world outside the machine or about a productive programming process.

BRIOT: So what's the next step?

GASSER: Well, pretty quickly, programming technology developed into 'high-level' languages like Fortran and PL/1 and then to Algol and Pascal. These languages developed control constructs like if-then, or loops, or block structures, and they had data structure constructs like variables, block statements, arrays, and so on. The critical point, *vis à vis* a progression toward agents, is that these 'higher-level' mechanisms use knowledge about the world and about effective programming practice to pretty much eliminate some programming possibilities and make others a whole lot easier. So, again, that has two consequences: it helps manage complexity, and it links descriptions more closely to what's being described. Because of this, people can program more easily in higher-level languages. Of course there's been plenty of debate over what's right, which language is more expressive of what kind of functionality, which restrictions work and which don't, and so on. Still, in general, these technologies have been progressively moving toward managing complexity for programmers and maintainers by embedding more structure and contextual knowledge. Some of the same debates are now playing out in agents research---what interaction languages should agents use, what development frameworks, what architectures, and so on---even "what's an agent?". They just aren't settled yet---the experiments are still going on.

In the 1970s Structured Programming brought programming methodology---people's work---into the picture and coupled it with programming languages and tools. This is another, even stronger link between programming technology and the outside world. There were many debates and specific, conscious decisions about what parts of tasks and programming methods should actually be included in programming languages themselves or supported by software environments or software artifacts. This time, too, the general conclusion was that it's extremely important to place certain constraining structures and interpretations of the world into the realm of programming languages, the programming process and the resulting software artifacts. I'd call the underlying issue here 'division of labor'---what goes into the tools and programs, and what is the province of people: developers, users, or whomever? Once again, the same questions are being raised about agents now: what should be in an agent, what should be in the environment, what should agent do, what should people do?

BRIOT: OK, I see that people added structures and knowledge to their programming tools over time to handle complexity. How does this actually progress up to agents in the historical sense?

GASSER: Over time it's kind of like the 'outside' and the 'inside' of programs and machines has gotten more integrated--more of the world has been brought into the domain of the machines, in the form of structures and conventions that have proven useful in the world. People started to look seriously at distributed objects and object-based concurrent programming (OBCP) back in the early 1980's, partly because the environment for programs had changed so much with networks and distributed machines. OBCP and distributed computing languages started to manage concurrency in a structured way that was built right into the OBCP languages, not just built outside in the operating system using semaphores or mutual exclusion primitives. For example, the first versions of Concurrent Smalltalk in the early 1980's were built by starting with Smalltalk and reengineering it to accommodate concurrency in a clear way. In a certain sense the problem was to combine the two intellectual threads of 'structures for managing concurrency' that had been developed in operating systems and concurrency theory, with 'structures for managing description and action' that had been so highly developed in object-oriented programming languages and language theory in general. Distributed objects try to introduce some very formal mechanisms to manage the complexity and uncertainty that concurrency introduces. Just as before, it's always a battle between free, open complexity and the need to control that complexity with structure---in this case with structure like concurrency mechanisms. This is one of those cross-disciplinary unifying syntheses that we were talking about earlier, that had a big impact. Of course, the extreme is the case of actors---the idea was to extend the structure of concurrency and objects down to the very roots of the whole approach, so that instead of 'managing' concurrency, as though it were a problem, you have to 'manage' sequentiality as a special case. But I think we'll say more about Actors later.

BRIOT: Yes, I hope so. At this point, though, can you say more about how you think the context around programming technology was brought into the picture, and how this is related to agents?

GASSER: Sure. The key recent step was the concept of subroutine libraries. All the programming technologies we've been talking about through this historical picture, even OBCP languages, were essentially domain independent to this point in history---the mid-1980s. They didn't deal at all with the real concrete semantics of the domains in which programs were written. And of course the fact that that we could address so many problems of structure and control while remaining so domain independent is a great benefit and a terrific intellectual and practical achievement!

But as object-based and OBCP systems got built and used in practical environments, a lot of focus turned to subroutine libraries, and later to the so-called 'components.' Again, it was too hard to keep building all the low-level detail for each specific application every time. Programmers needed to capture aspects of the domains and reuse them, and they turned to libraries, which are extensions to programming languages. So C++, for example, relies heavily on a massive construction of libraries. Smalltalk relies on a massive construction of libraries. Java relies on a massive construction of libraries. What are these libraries? Well, these libraries are collections of information that is relatively more domain dependent and that specialize and conventionalize ways of programming for specific kinds of problems. So you have libraries of user interface structures, like windows, widgets, and text representations---which are built on particular theories of human-computer interaction. Or, you have libraries of database interaction tools---SQL query interfaces for Java, or Smalltalk, or whatever. These depend then on the development and structure and assumptions of SQL, and also on the use and structuring of data bases which are coherent with SQL.

BRIOT: OK, I agree, it seems these days that people generally see libraries as a critical part of modern programming languages, and this certainly wasn't the case 30 years ago. But how do libraries relate to agents?

GASSER: Suppose we sum it up this way. Libraries were yet another step in the long-term process of building parts of the outside context and ways of managing complexity directly into the languages and tools we use, and agents are part of the next step. So historically, we're at a point where we can talk about agents not as something mystical or anthropomorphic, but instead as a contribution to the progression of programming technology. Agents are the next step in this long-standing two-part process.

BRIOT: So of course the next question is what specifically does the idea of 'agent' add to this progression at this point in time?

GASSER: Yes. Here I think the critical insight about agents---the next step in the progression of adding structure to programming technology---is the idea of direct support for 'structured persistent action.' People talk about this as autonomy, they talk about this as goal-directed behavior, or whatever. But for me it's the concept of structured persistent action. In effect, ideally what the idea of agents gives, is a new programming tool, along the lines of objects, and along the lines of concurrent objects, but now a bit farther along in terms of the control capabilities built into the tool or framework itself. With agents, persistent action becomes part of the toolkit--part of the framework that the programmer doesn't have to build explicitly from scratch.

BRIOT: What about the other aspects that are typically included in the idea of 'agenthood'---world models, the ability to communicate, deliberative, conditional, or reactive action, and so on?

GASSER: I may get into a religious war here, but most of these don't seem to me to be critical or fundamental advances in programming methodology. Some of them, like communication or deliberation, are already clearly the capability of objects or even subroutines. Adding the overlay of talking about them as attributes of 'agents' doesn't do much to make 'agent' a unique concept. What's the value added, especially from a tangible system-building perspective?

BRIOT: So you think that 'structured persistent action' is a programming technology that both differentiates agents and is also an added value.

GASSER: Yes. You can now have a program that will persistently try to accomplish something, somewhat irrespective of what you do to program it. It comes with a little kind of persistence engine. My young son Liam has a book called "The Little Engine That Could." This book is about a little steam engine, a train, that took up the goal of pulling a load of toys over a hill, and that continuously tried to get there, and never gave up, never failed---it's a kid's lesson in self-reliance and perseverance. OK. Rather than the programmer controlling the failure and success of a module or component through the semantics of the program, with the structured persistent action concept you get to build a little capsule that continuously strives for success without accounting at the outset for all the possible conditions it will face or in what order. What you provide is a goal or a description of what success is, and maybe some collection of alternative methods or building blocks for getting there, and in the most advanced case, maybe even a way to learn or remember new methods when you need them. But the agent's orientation is to provide continuous seeking for that success, continuous persistence, with its own internal control regime.

BRIOT: How does this link to the idea of a progression in programming technology?

GASSER: It's as though given the recent state of experimentation and knowledge of programming, the community has said now, at this point, the next level of structure we want to add to our tools is a self-directed behavioral structure, namely 'structured persistent action.' Of course, just as before, it's currently just a hypothesis that this will be a useful, usable and productive thing to do!

BRIOT: How do agents really enact this idea of persistent action? What does it take to get the persistence?

GASSER: Achieving it has some requirements. I think it first takes a locus of control, a continuous operating engine. Happily, we already have this with the concepts of distributed objects and actors. But it also takes the idea of choice and the idea of options. So, rather than there being only one way of accomplishing something, that is to say an algorithm which follows a strict and predefined path, the proper concept of an agent implicitly includes both a repertoire of options or possible next steps, and the ability to choose among them at the time of action, in the actual context of action, not at the time of programming. So the idea of choice at the time of action, rather than choice at the time of programming, is a key notion. That means three aspects of an agent are critical: number one, it has to have 'knowledge' of what is possible. That is to say, it has to have the domain over which to choose, it has to have a range of options. Number two, it has to have some interaction with the world around it, which is going to give it feedback about its choice, whether it is successful or not. And number three, it has to have a control structure that is primarily iterative--that can repeat conditionally. Abstractly, it's 'repeat-until' at some level.

BRIOT: Does this give us an idea of some possible agent architectures, at least at an abstract level?

GASSER: Yes, exactly. In general, the underlying concept of an agent has to be something like search to accommodate this persistence of action capability. Search is one of the most interesting ways of dealing with openness---with an unpredictable environment. Search is so fundamental and interesting because it is the one boundary at which openness, unpredictability, and algorithms come together. Search is an algorithm for dealing with fundamentally unknowable situations. The 'test' or 'evaluate' component of search is what makes this possible, and that's why persistent agents have to have this component of interaction with the world and interpreting its feedback.

Whether an agent is implemented using a specific search algorithm or not, I'm not concerned about. But the fact is, the overall iterative control structure, the set of options, the choice at the time when an action occurs, and some feedback as to the success of the choice---these things begin to define structured persistence of action. If the choices are successful, then the agent can proceed, if the choices are unsuccessful, then the agent has to be persistent, reorganize itself, choose another mechanism, and try again.

BRIOT: How do you see this in terms of the historical progression of programming technology we were talking about?

GASSER: Well, I think just a few ideas are important: first, there is a new programming paradigm that includes built-in persistent action. Second, the persistence is realized with a domain of possibilities, a choice mechanism, some feedback from the world that validates that choice, and an iterative control structure. And those, I think, are the key elements of the ideal or potential contribution of agents as a programming paradigm. But it's not really being addressed that way, which is why I think of it as a potential or an ideal. All the talk about tools like Agent-TCL, Java---most if not all of the proliferating 'agent building frameworks' currently available---ignore this issue of persistence of action and the basic requirement for a search-oriented approach. It seems to me that what many current agent frameworks do is already available as distributed objects---there's no really fundamental advance and probably not a quantum leap of added value for system building either.

BRIOT: Another issue you mentioned above was how over time programming technology increasingly includes world knowledge. How does that fit in here with agents?

GASSER: One other observation about persistence is that persistence requires this collection of choices for action. And the choices in the collection have to be relevant to the arena where action is taking place--they're domain-dependent. That means that the agent in a certain sense has to encode a lot more knowledge about its domain---agents are more tightly bound to their domains. We can see how the notion of persistence of action as a built-in capability is another step in a progression toward more refined, constrained, well-understood, specific and limited control structures. But it's also a progression along the path of adding more and more knowledge and structure from the world and from the operative context into the system itself to make the system more capable and more active in that context. In order to have a domain of choice, the domain of choice has to be relevant to the world where the choice is being made, that means the agent has to have a lot of encoded knowledge about what are the possible choices, how they make sense, when they work and when they fail. So an agent built in this way evaluates its success and its failure, and this kind of agent becomes more knowledge-based. I don't mean 'knowledge-based' in the sense of having structured knowledge bases and frames and objects and rules. I mean it simply in the sense of having a lot of knowledge about what are the options for action, and being structured so as to reflect the priorities of a domain of action. The choices that are given to an agent have to be related to the domain where it's working, because otherwise they don't make any sense, obviously. Along with options for action there is additional knowledge in the decision procedures---which ones are rational or make sense, in the face of a particular range of choices in some particular context?

So I think that what is exciting about agents, from a single agent perspective, that is this programming progression towards in these several directions---from this perspective agents can really be a progression in a historical process, not just a new buzzword. But it will take still some more careful work and experimentation and research.

BRIOT: Let's talk about some other issues that relate to agents and concurrency---for example actors, or mobility. How important are these for thinking about agents?

GASSER: Mobile agents raise some interesting problems of concurrency and interdependency because of the problems of asynchrony they introduce. As an agent moves from one place to another, it has to interrupt its activity, move, and re-start. But in the meantime, the world may have moved on. I just moved my family from Washington, DC to Illinois, and believe me I know the feeling! Here I suspect the actor model has some interesting application. Actors are inherently asynchronous, and the actor theories don't depend on synchronous action, as long as message ordering is preserved, which is pretty easy to do. Since actors are an inherently abstract idea that can be applied to objects of many granularities, largely independent of their internal architecture, actor theory and actor technologies may be useful tools. The basic emphasis on concurrency is also a very interesting orientation. And recent work in actor computation is also in line with our earlier discussion on increasing structure and convention over time as a way to manage complexity. Some of the recent work of Gul Agha's team has looked into how patterns in actor programs can be recognized at a low level, and even compiled out of high-level specifications. You know, one of my favorite ways of defining 'organization' is 'pattern of activity.' So lots of questions like what are the appropriate organizations or patterns for particular circumstances, how do patterns evolve and change rationally, how does macro behavior emerge from micro-structure and vice-versa---all these questions are important for evolving actor computations as well as for agents and multi-agent systems. I've always felt that asking questions like this can really begin to link several streams together from OBCP, Distributed AI/Multi-Agent Systems, and even organization theory.

BRIOT: Do you have any sense about how to compare actors and agents?

GASSER: At the individual level, I think actor theory and technology provide a lot of nice programming machinery. You can build agents out of actor technology, including even the sophisticated search-based agents that we talked about above. At the multi-actor or multi-agent level, the two begin to diverge though. This doesn't have to be the case, but it just seems that the communities are focused on different issues. For example, one of the main issues in multi-agent systems research is how to insure coherent coordinated behavior. Well, of course actors research is interested in that too! But one difference is that for MAS researchers coordination is an issue much more complex than concurrency control or resource allocation. One of the key, hard coordination problems in MAS is how to manage the uncertainty introduced by distributing a computation across agents. For example, several agents may be searching the same space for partial answers, but the conclusions they reach may be incompatible for the whole, if they're not coordinated. Because they're distributed, each agent may have tremendous uncertainty about the outcomes of the others' searches, and this is an impediment to efficient behavior individually and collectively. So agents need techniques that allow them to reason about what local or system-wide policies will allow them to reduce their uncertainty. To my knowledge, these ideas of managing uncertainty haven't yet been incorporated into OBCP technologies, and so there's an opportunity---there's clearly a gap that can be closed. But, as you know, this is just one illustration of how there hasn't been nearly enough good work linking actor concepts or other OBCP programming technologies with agents or multi-agent systems. It's a natural synthesis, of course, but the bridges between these two intellectual worlds haven't yet been built securely.

BRIOT: I agree with you that alot more can be done to link OBCP concepts and agent-based systems. Is there anything else about agent mobility that you think is important to discuss?

GASSER: Actually, yes. I think there's a set of issues that's much more fundamental and challenging than asynchrony and concurrency management, for mobile agents. Very generally, the issue is how can a mobile agent act robustly and predictably in an unknown context? The flip side is how can a system act robustly and predictably as it interacts with unknown mobile agents that arrive at its doorstep? This problem typically is cast as a security issue for mobile agents, and of course that's definitely a pressing issue. But I think it's a much bigger question than just security, and in fact has lots of fundamental implications and is intellectually very exciting. Just on the level of interaction languages, for example, there are plenty of basic questions about how tokens or utterances come to have useful interpretations in new, unforeseen contexts. This issue would apply to programming semantics as well as interaction language semantics, of course. I think this problem forces us to think about how agents can construct their interactions on the fly, in local contexts, rather than relying on pre-programmed interactions. And I think this on-the-fly construction of activity extends even to very basic foundations, such as agents reconstructing and evolving the languages they use to communicate, while they are communicating. You can't rely on pre-programmed solutions in an open world---that's sure---but aside from the search mechanisms we mentioned above, we have precious few foundations for dealing with openness, especially semantic openness. So I think this is one of the real intellectual challenges brought by mobile agents. It's not really new, but mobility makes it much more visible and pressing. It clearly hasn't been dealt with in terms of concurrent objects either.

BRIOT: Well, this conversation seems to have covered quite a lot of ground in the realm of agents, concurrent objects and many other issues. Anything else?

GASSER: Not really. We've just begun to scratch the surface of some of the basic issues in the Multi-Agent Systems (MAS) arena and how they may relate to collections of concurrent objects. These are very deep, very fundamental, and they may really change the picture again, even, I think, to the point of undermining a lot of what we think we already know about the nature of information and about computing. But that's another story too!

BRIOT: Well, maybe we'll leave that for another time. Thanks, Les.

GASSER: It's always a pleasure, Jean-Pierre.