What A BPMS Is

Howard Smith
Computer Sciences Corporation

It is important to distinguish marketing messages from the emergence of a new category of software. As I have reported before, vendors and analysts love creating new acronyms for technologies old and new; it seems to be the perpetual mark of the IT industry. On the other hand, when Ted Codd announced the characteristics of relational data management in his paper, “A Relational Mode of Data for Large Shared Data Banks,” published in the June 1970 issue of the *Communications of the ACM*, many knew instinctively that the way business data would be manipulated in computer systems was to change forever. And so it has.

Likewise, when Robin Milner published *Communicating and Mobile Systems: the Pi-Calculus*, in 1999, the door opened to a new way of writing software where computation and communication are no longer treated as two very different things, but as one new abstract type, the *process*. Such a model is sorely needed, for today computing is conducted on global networks that link teams, business units, companies, and even whole industries.

The relational model opened the door to a much broader, much more reliable manipulation of *data* by computers. It led to general purpose data manipulation languages such as SQL. Software, and its data were two different things, even if, as in object-oriented computing, data was kept close to the computing methods in an object class. The pi calculus has opened the door to a much broader manipulation of *processes* by computer systems. It will lead to a general purpose process manipulation language, to be called BPQL.

Relational data expert, Chris Date, has observed that, “The entire relational database industry, now worth many billions of dollars a year, owes the fact of its existence to Ted’s original work.” On the other hand, data management analyst Fabian Pascal wrote that, “Those who made fortunes from his invention shared nothing with Ted; not even that *Forbes* magazine published Larry Ellison’s photo next to its listing of the model among the 85 major innovations of the 60s - 70s decade.”

When Robin Milner published the pi calculus, he did not imagine the possibility of a Business Process Management System (BPMS). Only recently, with the work of the BPMI.org, has this possibility been demonstrated to be viable in practice. This is not to diminish in any way Milner’s work; far from it. Milner has demonstrated in numerous papers and lectures a computing paradigm where computer systems are built from communicating parts that form processes, rather than from adding in communication between parts as an extra level of activity. This idea is extremely significant in computer science and far beyond its relevance to the construction of a BPMS. A BPMS is just one new category of business system that can now be developed because of Milner’s work.

Milner would not equate his pi calculus model solely to the BPMS, any more than Codd would equate the relational algebra to the RDBMS. So arguments about the significance of the BPMS, compared to other platforms, imply nothing about the significance of these mathematical foundations. Pi calculus is deeply important. The pi calculus does not replace the relational algebra; however, it gives us a new understanding of how relational data contributes to processes. Previously, programmers only concerned themselves with how to manipulate data using code. Now, they can turn to thinking about how to manipulate processes, and clearly understand how processes change.

Because the relational model of data is not a storage-model does not prevent the RDBMS from reliably storing data; indeed, that is its function. The same is true of the pi calculus. It is not a storage model or an execution model, yet this does not prevent a BPMS from both persisting and executing, processes. Indeed, a BPMS may use a relational model to store the processes themselves, or it may choose not to.

One of the main purposes of the relational model was to separate the logical aspects of the data—what users and applications care about—from physical storage details. Likewise, the pi calculus separates the logical aspects of how participants in a process communicate from the actual means of communication. A BPMS can, according to the pi calculus, orchestrate processes occurring over networks using messaging, threads, shared memory or, within the domain of its own *process virtual machine*, it can implement the abstract primitives that form the pi calculus, with no messages required.

A BPMS is a concurrent system. Concurrency is an inherent part of the language used to build a BPMS, just as objects were inherent to Smalltalk. While concurrency can be used to extend the BPMS over networks to include other systems, this is not the only purpose for concurrency in a computer language. As users interact with systems, they too are concurrent participants. Thus, processes let us build more useable systems. Systems built atop databases with no process often feel, to users, as little more than glorified electronic filing cabinets and a thousand data entry screens. Concurrency is evident in the world around us, and, hence, concurrent languages make programming a whole lot easier for many real world problems. But even if the inherent concurrency within the BPMS is only used to drive orchestration of existing services, it still makes sense. Programming networks has just become a whole lot easier. No need to handle code and message independently; just sketch out a process, and, hey presto, it springs into life. What's code can be message, and what's message can be code, interchangeably. Data, in a world of process, is the state of the process. Process is the new data.

Relational model under scrutiny

Codd's death in 2003 came at a time when his mathematical database model was coming under the most pressure. His pioneering work has powered businesses around the globe, and created a structured and reliable data management infrastructure upon which we all depend. Indeed, those who predicted that the next generation of databases would be object databases have been proved wrong. Yet vendors who have built multi-billion dollar companies on Codd's set of mathematical principles have sought to advance towards a new world of computing—one based on XML—that can store and manage variously structured objects with the same ease that Codd allowed us to store and manage related data elements. Yet at the same time, as Fabian has observed, "Is it realistic to expect some 'great innovator' to achieve what is an extremely tall order: the replacement of predicate calculus (logic) as the foundation of database management?" Such foundations don't die easily. In a very real sense, they are immortal.

Now before I proceed from this point, it is extremely important that my readers understand that I am not proposing that the pi calculus is the replacement for Codd's relational algebra based on predicate logic. Bear with me for the next few paragraphs, and you will sense a great opportunity.

It is interesting to note that, as well as objects and the flexible structures of XML, other models have been proposed that challenge the relational model. The best known is the associative model, proposed by Simon Williams.

Proponents of the associative model point out that the most commonly touted limitation of the relational model, an inability to describe unstructured data, is not a limitation at all. The only form of data that could legitimately fall outside of the expressive power of the relational model would be random (noise) data. Rather, proponents of the associative model point to other, less commonly understood, deficiencies of the relational model, and therefore of relational database management systems. These less well-known deficiencies of the RDBMS, they point out, are nevertheless the cause of much difficulty in using database management in practice. They point to the complexity of today's data models that lie beneath the immense quantities of code that form today's bloated packaged application suites, including, ERP.

It has been noted by users of relational database systems, and by proponents of the associative model, that every relational application needs a new set of programs developed from scratch, which is labor-intensive, expensive, and wasteful. True enough. While a relational database isolates programmers from the intricacies of the physical storage of data, it does not always isolate them from the logical structure unless great care is taken in data model design. Indeed, software package vendors, most notably ERP vendors, have developed an approach for dealing with this. Each vendor of ERP has found it necessary to add a layer of data abstraction above the relational model so that they can more easily write and maintain diverse software modules that, nevertheless, manage and share consistent data. Put another way, there is nothing in the relational model per se that helps programmers cope with logical data model independence.

Similarly, proponents of the associative model point out that relational applications cannot be readily customized to the needs of a large number of individual users. True again. There is nothing in the relational model that serves this purpose. Associate model proponents also point to the fact that the relational model cannot record a piece of information about an individual thing (entity) that is not relevant to every other thing of the same type. Proponents of the associative model also point out that, because of lack of associations in the relational model, information about identical "things" in the real world is structured differently in every relational database management system.

These "deficiencies" of the relational model, and of RDBMS products, have real world consequences according to the proponents of competing models for data such as objects, associations, and XML structures. For example, while application vendors may add logical abstraction layers to their databases to counteract the deficiencies, each does so in a different way, leading to complex application integration at run time across multiple systems. This "deficiency" in the relational model has spawned the need for an entire sub-industry, enterprise application integration (EAI) solutions. Similarly, if software developers create tables to handle customization for different users, they will each do so in a different way, further complicating software maintenance and integration among computing solutions.

Indeed, if we look in detail at the problems that beset packaged software implementations by end users and consulting firms, many difficulties can be traced back to deficiencies in the relational model. Everyone knows stories of how hard it is for ERP suites to be combined with other best-of-breed packaged applications. Everyone knows that ERP applications tend to proliferate in the enterprise for reasons that are, uh, *unclear*. Some organizations have hundreds of ERP instances. These appear most difficult, or impossible, to consolidate back to a single ERP instance, let alone upgrade Companies often give up on the exercise and divide processes between a core enterprise suite, usually limited to HR and finance functions, and leave other applications alone. If they do chose to base all applications on ERP they can be caught out by considerable costs. These arise from deficiencies in the ERP development environment, including, maintenance, upgrade, and customization.

Companies that wish to upgrade from one ERP version to the next or to switch to another ERP vendor face formidable challenges if the data abstractions added to the basic relational model and intended to isolate the software modules, do not support the upgrade path. Indeed, for those organizations that heavily customize ERP applications, the situation is worse, for under those circumstances, the ERP vendor cannot protect the end-user organization against itself by ensuring a clean and consistent data abstraction layer. Once that layer has been compromised through customization, life gets very tricky indeed for future maintenance. It is not uncommon for organizations to have tens or a hundred or more ERP instances. Such organizations can be faced with a future projected bill for ERP development activities of tens or hundreds of millions of dollars.

Proponents of the associative model might be forgiven, however, for believing that, if the whole world converted to associative databases as the foundation for computing, all would be well. And, indeed, much pain might be eased if this change occurred, although, I am equally sure that other problems, not foreseen today, may arise with the associative model. And we all know what happened to those who set out to define a single mega data model for the whole enterprise, hoping to solve all the data modeling in one go and provide a consistent model for all applications. The world is not that simple.

Yet, despite these problems with the RDBMS and applications built on it, there is no industry-wide consensus to discard the relational model. Not everyone has seen the goodness and light that proponents of associative data have seen. Other approaches have been proposed and developed to deal with today's deficiencies in the relational model. Objects, shared industry data models, semantics, and flexible XML structures are all attempts to improve the basic model of computing. After all, let us not forget that the majority of applications "out there" are, in reality, tightly coupled to the relational data models upon which they are built. It is not so easy to get rid of all this code. What should we make of this?

No shortcut via XML

There is much to like about the associative model of data. If it were to be implemented on top of the relational model as an abstraction layer, consistently among all package solution vendors, the world probably would be a better place. But unless database products enforce this in practice, it is unlikely to happen. The value of any model is maximized only when it is *implemented*. The fact that Codd's relational model was implemented within database products according to his rules meant that the developer using the database would obtain, without effort, the advantages of Codd's model, such as flexible data retrieval or update capabilities. Codd's Rule 7, for example, asserts that an RDBMS must support insert, update, and delete operations for any retrievable data set, rather than just for a single row of a single table.

While I am sure incumbent relational database vendors have looked at many innovations, including associative models, the primary characteristics of the relational model are still the bedrock of today's database systems, despite the availability of the associative model from vendors of associative databases, and despite the existence of object-based computing and XML. Associative database vendors may point out that today's applications, based as they are on the RDBMS, are costly to develop and maintain, and that the biggest driver of cost and complexity in packaged applications is the result of the underlying relational database. But this has made no practical difference. And this is not surprising, because Codd's work is founded on the predicate calculus. It is simply a great model for data, but it says nothing about the software built on top of that data. Herein lies its Achilles' heel.

The relational model says nothing about the combination of data and code, i.e., the processes that arise as data is manipulated by code and transmitted through computing systems. Surely it is here where the problems lie, for data without application is nothing. In this respect, the associative model (of data) may be a real innovation in data, and may even lead to a degree of code reuse higher than that achieved using object computing, but, at its heart, associative data is still *data*. In the world of the associative database, code and data are still two separate and very different worlds.

And don't look towards XML for answers to the problems of the relational model and its awkward binding to code. XML is little more than a host-syntax for multiple forms of information. Essentially hierarchical, XML represents the very data model inherent in the early hierarchical data management systems that relational databases were developed to replace. XML, in reality the new ASCII code, may be infinitely extensible, but, like ASCII strings, it has no underlying model of computing. *XML is merely a container for other models:* relational, associative, and others. Neither objects, nor XML, contain any real math.

Towards a process solution

While Ted Codd was quick to point out that the relational model was not a solution to all possible problems or that it would not last forever, many believe the model can serve an extraordinarily large number of problems, and will endure for a very long time. Yet, at the same time, I urge those who challenge the relational model using objects, associations, or other flexible structures, to look again, and ask themselves deep questions about why they are asking for those ideas. Are the problems they observe in application development only the result of deficiencies in the relational model? I think not. If this were the case, software would not need to exist. The problem is not just about the way we represent data, but about how we separate, within our computing systems, the software that manipulates data from the data itself. Why do we do this? Simply because, prior to the pi calculus, there was no foundation for any other approach to work in practice.

It seems almost absurd to point out the obvious, that today we represent data and code in two different ways. Intuitively, we always think of data as something different from code, yet we accept that computer programs can read and write code as easily as they can read and write data. How else, for example, would we compile programs into lower level assembler code? But what if these two very different things, data and code, could be unified, not just combined, into one new kind of information, the *process*? Wouldn't that change the way we could program computers? This possibility is precisely what Milner's pi calculus provides.

A BPMS is inspired by the pi calculus

When objects were first announced, there was a lot of excitement in the technical community about the possibilities enabled by putting data and its code together in the object. For example, inheritance, specialization, and reuse seemed a more natural way of programming the world, the view at the time being that the world was made of objects. But, in reality, data and code, while co-located in the object, were still two very different things. The grouping of object and code, while convenient for modeling purposes, in no way meant that data and code were anything other than they already were. In the pi calculus, by contrast, all that exists is a process. *A process is neither a piece of data, nor a piece of code.* It is something new. And advocates of the new process-oriented programming, or process design on a BPMS, think of the world as comprising processes, not objects.

A long, long, time ago, logicians came up with the idea of a Turing machine. Its basic activity consists of reading and writing onto a storage medium, such as a long piece of tape, a set of registers, or read/write memory. Today's modern imperative computing languages are built on this foundation. Logicians also created the Lambda calculus, on which the notion of parameterized procedures is formed. For example, in a modern imperative language, procedures can be written in advance, and then invoked with actual parameters. None of these models deal with the situation when two computational threads communicate across an interface with a handshake, which means that two otherwise autonomous entities have synchronized an action.

What Milner achieved with the pi calculus was a language whose expressions could describe these interactive processes. The pi calculus is the basic math to meet the challenge of unifying the way mobile distributed processes are defined, and the way they are programmed. Put another way, we now have a way to understand complex processes that involve interactions between many participants, in the same way that we have a way to understand computation with data in a single thread. What this means in practice is that it is possible to achieve a concurrent computer system in which the way it is programmed and the way it executes are one and the same. One thinks immediately of two or more programs interacting, of two computers on the Internet exchanging information, and messages being passed. And while messages may be the means of communicating across an interface, the message is not the process, merely evidence that the process is executing. With BPMS, we have a process machine, not a data and code machine.

In Milner's 1999 book, *Communicating and Mobile Systems: the Pi Calculus*, he shows many examples of such processes--the movement of a piece of data inside a computer, the transfer of a message, or, indeed, an entire program, over the Internet, clicking on a hyperlink, a job scheduler, data structures and the behavior of object systems: all examples of his general notion of a process. Indeed, theoretically speaking, even lowly numbers or simple calculations can be considered as processes. *The pi calculus is the means of symbolically representing, or writing down, those processes.* What Milner has found is a powerful generalization that, at a stroke, unifies many aspects of computing that we previously thought of as quite different in character. In fact, every program that ever existed, and all the ways in which these programs can interact, can be written down in the pi calculus using just a small number of symbols.

Milner, an ACM Turing Award winner, has demonstrated in numerous research papers and lectures, the power of the pi calculus to enable this new understanding of informatics. And, just as Codd's relational algebra gave rise to the development of a new platform for computing, the relational database management system, the pi calculus has inspired innovators to create a process management system, the BPMS.

At this point in my short paper it should not have escaped your attention that this new *process* raises the possibility of removing the problems that beset software developers when they develop applications on top of a data foundation. As we saw, these problems arise because of the coupling between computation code and the data models upon which it depends. These problems arise whether the data is relational, associative, or some other data-formalism. What's wrong is not the relational model of data, but, rather, the fact that data and code are separated. Processes unify, and therefore, eradicate, the problem.

What is a BPMS?

A BPMS is a new computing platform built on processes, not separate code and data.

For commercial business systems, there is an analogy between an RDBMS and a BPMS. *Data is persistent in the RDBMS. Processes are persistent in the BPMS.* An RDBMS needs to be reliable if it is to manage critical business data. A BPMS needs to be reliable if it is to manage critical business processes. *An RDBMS can be extended with software programs that manipulate data, so it is a platform for business applications. A BPMS can be extended with software programs that manipulate processes, so it is also a platform for business applications.* For example, a process design tool, a process analysis, or query tool, etc., is an application *of* the BPMS. An RDBMS supports data-changing transactions. A BPMS supports process-changing transactions. An RDBMS can aggregate data from multiple sources. A BPMS can aggregate processes from multiple sources, particularly, existing IT systems.

But look inside a BPMS, and similarities with the RDBMS quickly vanish. For example, processes, unlike data, are not static until changed by code; they are active, executing entities. *Processes are the code, and the data.* So, the BPMS must contain a process virtual machine that progresses, or executes, *processes*, according to their design.

A BPMS is therefore a new way to write software, just as a spreadsheet was a new way to do calculations without resorting to programming. Of course, writing software on a BPMS is very different from writing software using standard Java or a proprietary language such as Oracle's PL/SQL. In fact, a BPMS implies a completely new type of programming language and a new way of programming. It is called process-oriented programming, and it has concurrency, which is inherent to all processes, as a built-in feature.

All other programming languages in common business use have relied, for concurrency, on mechanisms outside of the language. For example, Java programmers use a Java thread library, or calls to the operating system to fork a new operating system process (not to be confused with a pi calculus *process*). Whereas, *on the BPMS, everything is a process, and everything happens concurrently.* This means a machine able, for the first time, to represent processes in the broadest sense, including the operating system processes that, before pi calculus, we could see, but not describe. The first industrial strength language to do this is called the Business Process Modeling Language (BPML). It has that name because writing software on a BPMS is more like *modeling* data on a RDBMS than like writing Java code. Writing BPML is like laying out the schema for how the participants in the process, themselves processes, will interact when they communicate (execute).

Business came to enjoy the advantages of a persistent, reliable, transactional store for aggregated data. Businesses will come to enjoy the advantages of a persistent, reliable, transactional store for aggregated processes. Both offer a platform for business, the ability to use computer tools to manipulate critical business assets.

Developing a data schema is quite different from developing software code. For one thing, a data schema can be deployed on the RDBMS in one step, without distortion and without translation to another language. In the same way, developing a process schema is quite different from developing software code. It too can be deployed in a single step, what BPMS vendors call “zero code development.” Its sounds odd to be able to do this for process, since we are each steeped in expectations about code, including its development or generation. But a BPMS need not generate a line of code. *Whereas a data model on an RDBMS can never be a program, a process schema is an executable system and creates the same effect as a software program, with one click deployment.*

BPMS potential

While a BPMS won’t be used for all software development, far from it, it will be used for commonly needed *business processes.* Most discrete computing processes will eventually be shaped on the BPMS anvil, and software will not be the same again.

When the general purpose relational database was created, businesses created all kinds of data models to make the new capability available to them. Before this, technicians fiddled with complex hierarchical data formats or specialized flat file formats. When the general purpose spreadsheet was created, businesses created all kinds of numerical models to make use of the new tool. Before this, technicians fiddled with complex COBOL programs, and business users had to make do with the limited number of calculations (reports, etc.) the technicians could create, since the technicians were a scarce resource and a bottleneck. Today, we have a limited number of packaged applications, atop a standard relational database. Packages built on RDBMS are the new bottleneck. The BPMS breaks that bottleneck. With the BPMS innovation, business people will be set free to forge any number of processes they want, built atop a standard BPMS.

Early case studies show advantages over existing software development techniques when using a BPMS. Users report a much reduced process design to deployment time and resource. Compared to typical industry-standard ERP development practices, for example, a BPMS can reduce development times by up to 75%, development costs by up to 90%, and associated integration costs to other packaged solutions by up to 85%. These figures are just a start. Process design and deployment in a world of BPMS feels more like data design and deployment in a world of RDBMS, than it does working with, say, a modern Java- or C#-based IDE. It’s much more productive.

As well as productivity advantages, the BPMS can resolve some of the problems associated with relational models, as the associative-model advocates have highlighted. For example, BPMS-based applications share a *process* model, not just the data model, greatly increasing integration, reuse, and customization. In effect, “programmers” (I would rather call them process designers) are able to concentrate on the intent of the process design, unencumbered by having to specify details of how each of the computing elements should work together to achieve the required result. For example, the majority of traditional software programs have to keep track of some aspects of what happened in the past. Additional data tables are set up to record this. But the BPMS includes this function, so it does not need to be programmed in every process. That’s just the way processes work. Of course, they have a past, and the process is persistent, including its past.

It is not surprising then that ERP vendors are interested in moving away from building their applications only on a RDBMS, and towards utilizing a BPMS. Just as the associative database advocates urge, this will allow them to avoid arbitrary layers of data abstraction above the relational model. Enterprises will be able to build shared process models, not just shared data models. They will do this for the same reasons they used an RDBMS, consistency. As a result, process interoperability among packages will increase, and, eventually, the days when businesses struggle with waves of multiple ERP instances, customization, consolidation, and upgrades will be a thing of the past.

This will take years, of course. In the meantime, leading companies will test the BPMS for individual applications, i.e., specific processes, as was the case for early RDBMS projects. And, incumbent ERP vendors will, along the way, battle to control the BPMS innovation, even to the point of trying to develop a BPMS on their own, risking a non-standard BPMS in each and every vendor’s application suite. Let’s hope that Robin Milner’s science, and clean languages developed from it such as BPML, pave the way for a standards-based BPMS, just as we have standard RDBMS products today. Why is this so important?

End users don't want more BPM standards arranged in a complex technology stack of components that then have to be integrated. End users want a standards-based BPMS they can acquire, install, and put to work. And the standards upon which such a BPMS is based ought to be theoretically sound, as was Ted Codd's relational algebra based on predicate logic. I can think of no better foundation for the BPMS than Milner's pi calculus.

The Road Ahead

At some point in the past, Ted Codd's relational model came to life as an RDBMS. Today, Milner's pi calculus model has come to life as a BPMS. What the BPMS does is to ensure that all applications (*processes*) developed on it, adhere to a sound model of process. We learnt that was important for data query, and Codd's relational model is still with us today, despite attempts to unseat it. We are learning we need the same for processes, and I have no doubt Milner's pi calculus, and Codd's relational algebra, will still be with us a century from now. I believe this not because I advocate a BPMS, but because of the utility of the model and the strength of the maths upon which it is based. The process model of computing just makes common sense—and, Milner's pi calculus makes the magic work. More than any magic lines of code, this is what a BPMS is all about: the very new, humble, but mighty, abstract data type, the *process*.