Logic Programming for Industrial Software Engineering*

Kung-Kiu Lau[1] and Michel Vanden Bossche[2]

[1] Department of Computer Science, University of Manchester
Manchester M13 9PL, United Kingdom
kung-kiu@cs.man.ac.uk
[2] Mission Critical
Drève Richelle, 161 Bat. N, 1410 Waterloo, Belgium
mvb@missioncriticalit.com

Abstract. Current trends in Software Engineering and developments in Logic Programming lead us to believe that there will be an opportunity for Logic Programming to make a breakthrough in Software Engineering. In this paper, we explain how this has arisen, and justify our belief with a real-life application. Above all, we invite fellow workers to take up the challenge that the opportunity offers.

1 Introduction

It is fair to say that hitherto Logic Programming (LP) has hardly made any impact on Software Engineering (SE) in the real world. Indeed it is no exaggeration to say that LP has missed the SE boat big time! However, we have good reasons to believe that current trends in SE, together with developments in LP, are offering a second chance for LP to make a breakthrough in SE. In this application paper, we explain how this situation has arisen, and issue a "call to arms" to fellow LP workers in both industry and academia to take up the challenge and not miss the SE boat a second time!

2 The Past

Before we explain the current situation in SE, it is instructive to take a brief retrospective look at both SE and LP.

2.1 SE: The Software Crisis

SE has been plagued by the *software crisis* even before the term was coined at the 1968 NATO Conference on Software Engineering at Garmisch. Despite progress

* This paper also appears under the title 'Logic Programming for Software Engineering: A Second Chance', in P.J. Stuckey, editor, *Proceedings of the Eighteenth International Conference on Logic Programming*, *Lecture Notes in Computer Science* 2401:437-451, Springer-Verlag, 2002.

from structured or modular to object-oriented methodologies, the crisis persists today. As a result, software is not trusted by its users. At the European Commission workshop on Information Society Technologies, 23 May 2000, an invited expert from a major microelectronics company stated that "major advances in microelectronics increase the pressure on software, but the fundamental problem is that we don't trust software".

So-called Formal Methods, e.g. VDM [15], Z [27] and B [2], were introduced to address the issue of software correctness. Whilst these have been successfully applied to several safety-critical projects, their practical applicability has been limited due to the high cost they incur. Additionally, there is the problem of "impedance mismatch" between a mathematical specification and an implementation based on traditional imperative languages such as C, Ada, etc.

2.2 LP: Unexplored Potential for SE

Like any declarative language, LP languages like Prolog can offer much to alleviate the software crisis. In particular, they can address software *correctness*.

A theoretically sound declarative language allows:

(i) the construction of a purely logical/functional version of the program based on a clear declarative semantics; and
(ii) the transformation into an efficient program.

With commonly used programming languages, correctness is hard to obtain (and to prove) whereas high-level declarative languages support and nurture correctness.

Software correctness, or the lack of it, is of course at the heart of the software crisis. So LP would seem to have the potential to make an important contribution to alleviating the software crisis. However, in the past, Prolog (or any other declarative language) was never seriously applied to SE in industry. This may be due to various factors, e.g. it may be because Prolog did not have the necessary features for programming-in-the-large, or it maybe because Prolog, or even the whole LP community, was not motivated by SE, and so on. Anyway, whatever the reasons (or circumstances), the consequence was that the potential of LP for SE has not been properly explored hitherto.

2.3 SE and LP: The Integration Barrier

Not even the staunchest LP supporter would claim that LP could compete on equal terms with the traditional imperative paradigm, especially OO Programming (OOP), for SE applications in general. So it is not realistic to expect LP to take over completely from the imperative paradigm that is predominant in SE. Rather, the only realistic goal is for LP to co-exist alongside the latter.

We believe LP's role in this co-existence is to address the critical kernel of a software system, for which there is no doubt that LP would be superior (for the reason that LP can deliver software correctness, as explained in Section 2.2). It

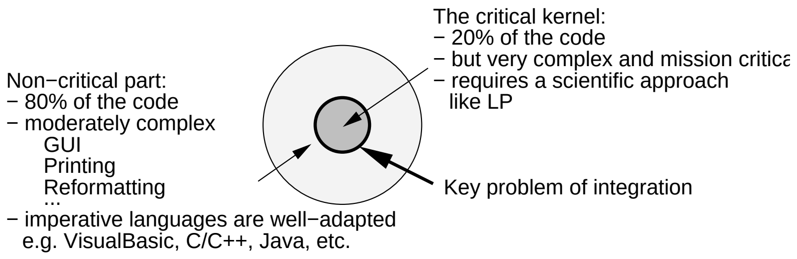
The critical kernel:
− 20% of the code
− but very complex and mission critica
− requires a scientific approach like LP
Non−critical part:
− 80% of the code
− moderately complex
GUI
Printing
Reformatting
...
− imperative languages are well−adapted e.g. VisualBasic, C/C++, Java, etc.
Key problem of integration

Fig. 1. The integration barrier between LP and predominant paradigms in SE.

is generally accepted that the critical kernel of a software system usually consists of some 20% of the code (see Figure 1), and it is this code that needs a scientific approach such as LP affords. However, even if LP was used for the critical code, the problem of integrating (critical) LP code with (non-critical) code in the predominant (imperative) languages would at best be difficult. For example, we could use a foreign function interface, usually in C, but this is often difficult. Thus, as also shown in Figure 1, there is an integration barrier between LP and the predominant paradigm in SE. This barrier would have to be overcome even if we were to use LP just for the critical kernel.

3 The Present

The software crisis persists today, despite the 'OO revolution'. LP has still not made any impact on SE. However, both areas show portentous movements.

3.1 SE: Dominated by Maintenance

Current industrial programming paradigms lack the sound and reliable formal basis necessary for tackling the inherent (and rapidly increasing) complexity of software, the extraordinary variability of the problem domains and the continuous pressure for changes. Consequently, current SE practice and cost are dominated by maintenance [11]. This is borne out by the many studies, e.g. [3], that strongly suggest that around 80% of all money spent on software goes into maintenance, of which 50% is corrective maintenance, and 50% adaptive (improvements based on changed behaviour) and perfective maintenance (improvements based on unchanged behaviour) [11]. This is illustrated in Figure 2 (taken from [11]).

3.2 SE: Moving to Components

Of course if software was more reliable, then the maintenance cost would decrease. Reuse of (reliable) software would reduce production cost. However, the level of reliability and reuse that has been achieved so far by the predominant OO approach is not significant. Large-scale reuse is still an elusive goal. It is therefore not surprising that today, with the Internet, and rapid advances in distributed

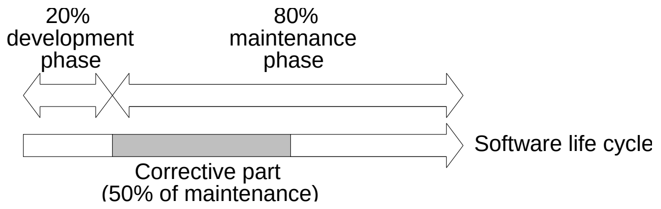

Fig. 2. Software cost is dominated by maintenance.

technology, SE is seeking to undergo a 'paradigm shift' to Component-based Software Development (CBD).

Building on the concepts of Object-Oriented Software Construction (e.g. [22]), CBD [29] aims to move SE into an 'industrial age', whereby software can be assembled from components, in the manner that hardware systems are constructed from kits of parts nowadays.

The ultimate goal of CBD is thus third-party assembly of independently produced software components. The assembly must be possible on any platform, which of course means that the components must be platform-independent.

The consequences are:

A Level Playing Field. CBD offers a level playing field to all paradigms. Current approaches to industrial SE cannot address all the needs of CBD, so the playing field is level and LP is not at any disadvantage.

A Fast Developing Component Technology. Component technology for supporting CBD is receiving a lot of industrial investment and is therefore developing fast. The technology at present consists of the three component standards CORBA [10, 4], COM [8] and EJB [28] supported by OMG, Microsoft and Sun respectively. Since by definition, it has to be platform and paradigm independent, this technology supports the level playing field.

These imply that CBD will overcome the integration barrier between LP and the predominant paradigm for SE, depicted in Figure 1. Thus CBD provides a realistic chance, for the first time, for LP to make a breakthrough in SE. We believe the importance of this cannot be overstated, and will devote a section (Section 3.4) to it.

3.3 LP: A Maturing Paradigm

In the meantime, LP has been maturing, as a paradigm for software development. Over the last ten years or so, the LOPSTR workshop series [19] has focused on program development. A theoretical framework has begun to emerge for the whole development process, and even tools have been implemented for analysis, verification and specialisation (see e.g. [1]).

A new logic-functional programming language, Mercury [26], has emerged that addresses the problems of large-scale program development, allowing modularity, separate compilation and numerous optimisation/time tradeoffs. It combines the clarity and expressiveness of declarative programming with advanced static analysis and error detection features. Furthermore, its highly optimised execution algorithm delivers efficiency close to conventional programming systems.[3]

So LP is in a good shape to take on the role of providing the 20% critical software as depicted in Figure 1.

3.4 SE and LP: CBD Overcomes the Integration Barrier

To reiterate, the crucial consequence of CBD, from LP's viewpoint, as mentioned in Section 3.2, is that component technology overcomes the integration barrier between LP and the predominant paradigm in SE. Therefore, we can update Figure 1 to Figure 3. This provides a realistic chance of a breakthrough for LP

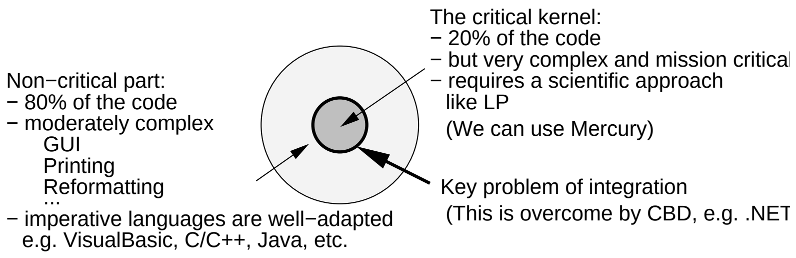
The critical kernel:
− 20% of the code
− but very complex and mission critical
− requires a scientific approach like LP
(We can use Mercury)
Non−critical part:
− 80% of the code
− moderately complex
GUI
Printing
Reformatting
...
− imperative languages are well−adapted e.g. VisualBasic, C/C++, Java, etc.
Key problem of integration
(This is overcome by CBD, e.g. .NET

Fig. 3. CBD overcomes the integration barrier between LP and SE

in SE.

We believe that a feasible practicable approach is to interface a suitable LP language, such as Mercury, to a current component technology. For example, we think that .NET [23], Microsoft's new component platform, could give LP the necessary component technology. As any language on a .NET platform can seamlessly interoperate with any other language on .NET (at least at a very low level), we have, for the first time, the possibility to build the critical components using the LP paradigm, while the non-critical, more mundane, components are still OOP-based.

This belief has propelled us at Mission Critical to invest in the "industrialisation" of Mercury [26], by interfacing it to .NET.

More specifically, we are working with the University of Melbourne on the following:

– integration with imperative languages through COM [8];
– multi-threading support [24];
– support for structure reuse, i.e. garbage collection at compile-time [20]: between 25% and 50% structure reuse has been observed in real-life programs;
– development of a suitable methodology which can guide developers who are confronted with a new programming paradigm;

[3] It is not our intention to engage in a 'language war' between Prolog and Mercury, or to debate our choice of Mercury.

– construction of a full .NET version of the Mercury compiler [9].

We have built a test Mercury.NET web service: a Coloured Petri Net component. Performance is very good, sometimes better than C#. There are still nitty gritty issues to solve (e.g. easier ways to produce the metadata related to an assembly), but they are being dealt with.

4 A Real-Life SE Application using LP

The results of the "industrialisation" of Mercury have enabled Mission Critical to successfully develop a real-life system, part of which uses Mercury. The system was developed for FOREM, the regional unemployment agency of Wallonia (in Belgium). FOREM (with 3000 staff and an annual budget of 250 million euros) is confronted with complex and changing regulations, which directly impact many of its business processes. After several contractors had failed to develop a satisfactory system capable of supporting a new employment programme, FOREM asked Mission Critical to develop such a system.

The requirements for the system were as follows:

– it should have a 3-tier architecture with a clean separation between the User Interface, the Business Logic and the Data Storage;
– it should be Internet-ready, i.e. it should have good performance and robust security when the user interacts with the services through the Internet or the FOREM intranet;
– it should allow easy modification of the business processes to cope with a continuously changing regulation.

Mission Critical successfully developed a system that met these requirements. The system, PFlow (Figure 4), is in fact the first ever industrial Mercury application. It has been in daily use since September 2000.

4.1 System Architecture

PFlow is based on the following design and implementation:

– business process modelling is based on extended Petri Nets (to leverage their formal semantics, graphical nature, expressiveness, vendor independence, etc.);
– data modelling is ontology driven;
– the client/server protocol is based on XML and the WfMC (Workflow Management Coalition) XML Bindings recommendations;
– a light client is developed in Java;
– a complex server is developed in Mercury;
– business calculations based on Excel worksheets driven by the Mercury application;
– component integration is done through COM.

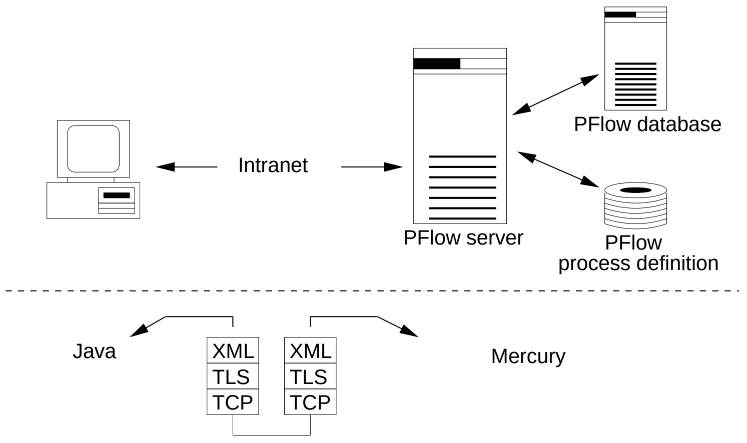

Fig. 4. Mission Critical's PFlow system.

The system architecture of PFlow is as shown in Figure 4. The main components of the system are a light client implemented in Java and a complex server developed in Mercury. The light Java client deals only with presentation issues, whereas the complex Mercury server provides the critical kernel of the system, providing a whole host of services including a Petri Net engine, folder management, alarms, business calculations, e-mail generation, transactions and persistence.

All state information in a Mercury program is threaded throughout the program as either predicate or function attribute values. To simplify the state information handling within the server a single structure, called *pstate*, encapsulates all relevant server state information. In the *pstate* structure, a general distinction has been made between values set at startup time (e.g. database names or SMTP server name and port number) and dynamic values which are constantly updated (e.g. folder cache, database connections). This distinction between the static and the dynamic state information means that options or customisation features can be added to modify the action of a part of the server with minimal effect on the other parts of the server.

The PFlow server operates in continuous loop accepting and processing requests as they arrive. When there are no outstanding client requests then the PFlow server uses the time to process any outstanding expired alarms, collect garbage, manage the cache and perform other internal housekeeping operations.

The protocol used between the clients and the PFlow server is a variant of the WfMC (Workflow Management Coalition) XML protocol. This protocol consists of several XML messages that must be sent to the server in a specific order (i.e. it is a stateful protocol) and the server must check the ordering. This protocol contains messages for creating a folder or a new task on a folder, finding tasks and updating a folder (and therefore its tasks, alarms, and dictionary entries). The message sequence is divided into two parts: *task identification* and specific *resource querying* and *updating*. Tasks are defined as a part of the process description (currently there are 12 main tasks, each of which might have a number of sub-actions or sub-tasks which have to be complete before they are removed from the list). The XML messages in the sequence required by the server are:

- `PropFind`, used to recover client initialisation information;
- `Create-Task`, used to indicate to the server that the client is interested in a specific task;
- `PropFind-Folder`, to search for a folder (or list of folders);
- `Create-Folder`, to begin the folder updating;
- `PropFind-Folder`, to recover the folder based on its identifier;
- `PropPatch-Folder`, to update a folder data item, e.g. an alarm, dictionary entry or task value;
- `Terminate-Folder`, to end the folder updating;
- `TerminateTask`, to close the current task.

When a client starts up, the first request is a `PropFind`. on the dictionary definition, and this is also used to clear any database locks or other information which might be associated with that client (so in the event of a client computer crash or untidy exit, the client can always be restarted).

4.2 System Evaluation

Profiling the server indicates that one third of the message processing time is spent in DBMS related operations. This means that optimisations in the user time almost get partly overshadowed by the database access and update operations. Additional strategies are being investigated to tackle this problem and reduce the impact.

Another problem has been the use of Excel as the 'business computing engine'. In a first version of the system, the server called Excel directly through the COM interface. The response time ($\sim$1sec to load the worksheet, send case data and retrieve derived value), although adequate for a prototype, was barely acceptable for a production system.

So, it was decided that the Mercury server would read the worksheet definition and use an internal Mercury representation. It would then interpret the Excel formulas internally (in Mercury) and keep the results as a Mercury data structure. This approach improved the response time considerably, reducing it to 30 msec (on a Pentium II, 350 MHz, 512 MB RAM), while keeping the standard Excel representation.

The current PFlow server has been used, in pilot mode, since March 2000, and in a full production environment, since September 2000, with currently 30 relatively intensive users across 3 sites. Since then the process description has evolved and been refined as requested by the customer. The system is being scaled up to 100 users working across 13 sites.

With the given number of users and sites, there are currently no performance problems and indeed quite the opposite since the work is being processed much faster – and much more reliably – using the PFlow system than with the paper based approach. However, there are known areas where throughput can be improved and bottlenecks eliminated, for example, making portions of the server multi-threaded or distributing the work across several computers, if demanded by further workload increases.

4.3 Appraisal of Mercury

Deploying Mercury in the PFlow system has re-affirmed Mercury's strengths for system development in general. The strict declarative semantics of Mercury means that side-effect based programming is not easily possible, so hidden program assumptions are obvious during development and the subsequent maintenance. Moreover, the combination of a strong type and mode analysis and module system with a declarative reading means a virtual elimination of certain classes of typical programming and development problems (e.g. memory access problems, incorrect function/predicate attributes, wrong types). Eliminating these problems means that the majority of the development time is spent where it should be – in solving the more interesting higher level conceptual problems, such as when to recompute an alarm or when to commit to the database any folder updates.

Furthermore, in Mercury, correct program development from specifications is simpler, and therefore less time-consuming, than in non-declarative languages. This is because in Mercury there is no 'semantic gap' between a logic specification and its implementation.[4] In our experience this has definitely been the case. However, the need for efficiency may necessitate transforming the simplest possible (and obviously correct) program to one that is not quite so simple but more efficient (and less obviously correct). Even here, any transformation technique employed, e.g. co-routining,[5] must preserve the declarative semantics, i.e. maintain the 'no semantic gap' scenario.

Of course in general there are classes of problems where a'semantic gap' does exist, in the sense that Mercury may not be appropriate at all. For example, for constraint-solving problems, a constraint logic programming language would be more appropriate.

Moving on to performance, for PFlow, Mercury has also delivered. The Java client in PFlow, although only dealing with presentation issues, requires 22,000 lines of Java code, whereas the Mercury server is developed with no more than 18,000 lines of Mercury code. This bodes well for Mercury's performance as far as cost (both production and maintenance) and reliability are concerned, by any accepted criteria, e.g. those in [11].

[4] For one thing, negation in Mercury is *always* sound because we have full instantiatedness information, so we never try and negate a goal that is not fully ground.

[5] Co-routining can be implemented in Mercury, but the programmer has to do it explicitly using the concurrency library, modelled on Concurrent Haskell [24].

4.4 Supporting Evidence for LP for SE

The success of this application supports our view that LP can be used for the 20% critical software, and more importantly that LP can be integrated with the predominant paradigm in SE by using CBD technology. Indeed, the language features of Mercury make it a superior choice for implementing the critical kernel of the FOREM application, the Petri Net engine. Our implementation of this engine illustrates this point.

We implemented the Petri Net engine so that it supports coloured Petri Nets, and will execute as a component on the .NET backend. Coloured Petri Nets [14] are typed. Each place in the Petri Net can only contain tokens with a specified type, as well as the arc expressions which determine what tokens are placed into the places. A goal of the implementation was to use arbitrary Mercury functions for the arc expressions and allow tokens which are arbitrary Mercury types. The expressive Mercury type system, in particular existential types [12], ensured that Petri Nets can only be constructed in a type safe manner, eliminating one class of bugs completely.

The Petri Net state also has to be serialisable so that it could be persisted in a database, if necessary. The Mercury type system also allows this by ensuring that each type to be stored in a place must be a member of a typeclass [12, 13] which can serialise and deserialise the type. Mercury's static type checking ensures that we can never construct a Petri Net which is not serialisable.

Finally Petri Nets are inherently non-deterministic. A transition fires when there exists a compatible token at each input place to a transition. This selection of tokens is modelled using committed choice non-determinism [21]. This allows the Mercury program to do the search to find the compatible tokens to consume, and then prune the choice point stack once a single solution is found. This gives us the benefits of the automatic search without paying the expense of the non-determinism once it is no longer needed.

5 The Future

We believe we are now at a critical juncture: our experience at Mission Critical has convinced us that LP has a chance to make a breakthrough in SE, but LP will only succeed if we collectively seize this opportunity in time.

5.1 SE: Component-based Software Development

In the foreseeable future, SE will increasingly emphasise CBD. The move to CBD is seen by many as inexorable. As we have seen, CBD has opened up the possibility of LP's co-existence with older paradigms at the moment. In future, we believe that in addition to (mere) co-existence with other paradigms, LP can play a crucial role in CBD as a whole.

At present the key pre-requisites for meeting the CBD goal of third-party assembly have not been met (see e.g. [6]), these being: (a) a *standard semantics* of components and component composition (and hence reuse); (b) good (*component*) *interface specifications*; and (c) good *assembly guide* for selecting the right components for building a specified system.

In [16–18] we argue and show that LP can play a crucial role in meeting these requirements. The cornerstone of our argument is that LP has a declarative semantics and that such a semantics is indispensable for meeting these prerequisites for CBD's success.

5.2 LP: Declarativeness Indispensable for CBD

So we believe that the role of LP in CBD is assured. The declarative nature of LP will increasingly come to the fore. As software gets more complex and networked, declarative concepts are increasingly recognised by industry as indispensable (see e.g. [5]). For example, declarative attributes are already common in security systems.

Industry are also beginning to realise and accept that declarativeness makes reasoning about the systems easier, and hence the systems are less likely to contain bugs. Even Microsoft is showing an interest in declarativeness. They have taken up the idea of expressing that a certain piece of executing code requires some constraints to be satisfied, and when these constraints are broken the system will refuse to execute. To reason about code containing constraints, of course you need a language with a simple semantics, e.g. a declarative one.

5.3 SE and LP: Predictable Software

Above all, the declarative nature of LP will enable it to be a key contributor to the ultimate goal for SE: *predictable software* built from *trusted components.*

In order for CBD to work, it is necessary to be able to reason about composition before it takes place. In other words, component assembly must be predictable; otherwise it will not be possible to have an assembly guide.

Consider Figure 5. Two components A and B each have their own interface and code. If the composition of A and B is C, can we determine or deduce the

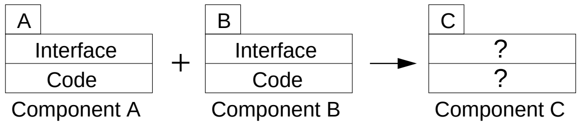

Fig. 5. Predicting component assembly.

interface and code of C from those of A and B? The answer lies in component certification.

Component Certification. Certification should say what a component does (in terms of its *context dependencies*) and should guarantee that it will do precisely this (for all contexts where its dependencies are satisfied). A certified component, i.e. its interface, should therefore be specified properly, and its code should be verified against its specification. Therefore, when using a certified component, we need only follow its interface. In contrast, we cannot trust the interface of an uncertified component, since it may not be specified properly and in any case we should not place any confidence in its code.

This is illustrated by Figure 6, where component A has been certified, so we

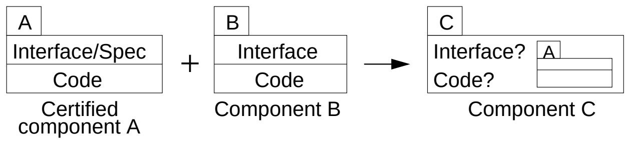

Fig. 6. Component Certification.

know how it will behave in the composite C.

However, we do not know how B will behave in C, since it is not certified. Consequently, we cannot expect to know C's interface and code from those of A and B, i.e. we cannot predict the result of the assembly of A and B.

System Prediction. For system prediction, obviously we need *all* constituent components to be certified. Moreover, for any pair of certified components A and B whose composition yields C: (a) *before* putting A and B together, we need to know what C will be; (b) and furthermore, we need to be able to certify C.

This is illustrated by Figure 7. The specification of C must be predictable

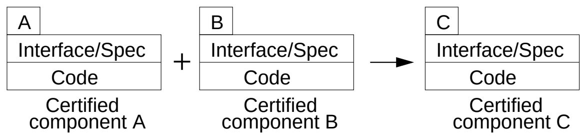

Fig. 7. System prediction.

prior to composition. Moreover, we need to know how to certify C properly, and thus how to use C in subsequent composition.

As an example of predictable software, one of the most interesting (and difficult) subjects today is *secure* software systems, systems that can be trusted. This is becoming a very hot commercial issue, especially in the context of web services built by a company A that could consume other web services built by companies B, C, etc. To be trustworthy, these systems must be *predictable*. In order to trust a system, we must be able to predict that the software: (a) will do what it is expected to do; (b) will not do what could be harmful; (c) or should something bad happen, will detect this and regain control.

Finally, the issue of predictable software, or predictable component assembly, is becoming more and more important, and we believe that LP should be able to make a key contribution here (see [16–18] for a discussion).

6 Discussion and Concluding Remarks

We have argued that CBD is giving LP a second chance to make an impact in real-world SE. Our belief stems from practical success of integrating LP to the traditional imperative paradigm via CBD technology, albeit using LP only for the critical kernel.

We have also stated our belief that in future, LP should be able to make a crucial contribution to the success of CBD as a whole. In particular, we believe that LP can be used to produce predictable software components.

Our sentiments here are very much echoed by voices on industrial forums such as CBDi Forum [7]:

> " ... the emphasis on well-formed components has diminished. This needs to be addressed and the necessity of good (trusted) component design communicated to all developers. ... "

> "Embrace formal component based specification and design approaches. Microsoft has already shown its interest in design by contract. This formal approach is a sensible basis for specification of trusted components.
> ... it is essential to understand and rigorously specify the behavior that the component or service, and its collaborations are required to adhere to. The conformance to behavioral specification is then a crucial part of a certification process which leads to trusted status.
> ... The challenge for Microsoft now is to provide support for delivery of trusted components and services, without reducing ease of use and productivity."

In addition, we also believe that logic and LP can be used for modelling and specifying software systems. The current standard of UML [25] has many limitations (not being formal enough), and we can do better than UML and have a completely formal logic-based modelling language.

Another interesting direction is "ontologies". The idea is to have a formal ontology describing problem domains. Logic and LP should be able to address the problem of the relation between the specifications and the ontology, the evolution of ontologies and specifications etc.

Finally, LP could aim at much more than a niche in SE. With the current rate of failures (Standish Group has observed that only 28% of projects are successful, i.e. 3 projects out of 4 have problems, and 1 in 4 is abandoned), a fundamental approach is needed. To borrow an engineering metaphor, you don't build a bridge with empiricism only (and debug it before you use it), you compute it first (with theories [mechanics], models [finite elements in the elastic domain], all "implemented" with mathematics). What we need is the same, i.e. the maths of software, discrete maths. LP is well grounded in this maths.

Acknowledgements

We are indebted to Peter Ross for his many helpful comments and points of information.

References

1. Theory and practice of logic programming. Special issue on Program Development, 2002.
2. J.R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.
3. R.S. Arnold. *On the Generation and Use of Quantitative Criteria for Assessing Software Maintenance Quality*. PhD thesis, University of Maryland, 1983.
4. BEA Systems *et al*. CORBA Components. Technical Report orbos/99-02-05, Object Management Group, 1999.
5. N. Benton. Pattern transfer: Bridging the gap between theory and practice. Invited talk at MathFIT Instructional Meeting on Recent Advances in Foundations for Concurrency, Imperial College, London, UK, 1998.
6. A.W. Brown and K.C. Wallnau. The current state of CBSE. *IEEE Software*, Sept/Oct 1998:37–46, 1998.
7. CBDi forum. http://www.cbdiforum.com.
8. The Component Object model Specification. Version 0.9, October 1995. http://www.microsoft.com/com/resources/comdocs.asp.
9. T. Dowd, F. Henderson, and P. Ross. Compiling Mercury to the .NET common language runtime. In *Proc. BABEL'01, 1st Int. Workshop on Multi-Language Infrastructure and Interoperability*, pages 70–85, 2001.
10. Object Management Group. The Common Object Request Broker: Architecture and specification Revision 2.2, February 1998.
11. L. Hatton. Does OO sync with how we think? *IEEE Software*, pages 46–54, May/June 1998.
12. D. Jeffrey. *Expressive Type Systems for Logic Programming Languages*. PhD thesis, University of Melbourne, Submitted.
13. D. Jeffrey et al. Type classes in Mercury. Technical Report 98/13, Dept of Computer Science, University of Melbourne, 1998.
14. K. Jenses. A brief introduction to coloured petri nets. In *Proc. TACAS97*, 1997.
15. C.B. Jones. *Systematic Software Development Using VDM*. Prentice Hall, second edition, 1990.
16. K.-K. Lau. The role of logic programming in next-generation component-based software development. In G. Gupta and I.V. Ramakrishnan, editors, *Proceedings of Workshop on Logic Programming and Software Enginering, London, UK, July 2000*.
17. K.-K. Lau and M. Ornaghi. A formal approach to software component specification. In D. Giannakopoulou, G.T. Leavens, and M. Sitaraman, editors, *Proceedings of Specification and Verification of Component-based Systems Workshop at OOPSLA2001*, pages 88–96, 2001. Tampa, USA, October 2001.
18. K.-K. Lau and M. Ornaghi. Logic for component-based software development. In A. Kakas and F. Sadri, editors, *Computational Logic: From Logic Programming into the Future*. Springer-Verlag, to appear 2002.
19. LOPSTR home page. http://www.cs.man.ac.uk/∼kung-kiu/lopstr/.

20. N. Mazur, P. Ross, G. Janssens, and M. Bruynooghe. Practical aspects for a working compile time garbage collection system for Mercury. In P. Codognet, editor, *Proc. 17th Int. Conf. on Logic Programming, LNCS 2237*, pages 105–119. Springer-Verlag, 2001.
21. Mercury reference manual. http://www.mercury.cs.mu.oz.au/information/documentation.html.
22. B. Meyer. *Object-oriented Software Construction*. Prentice-Hall, second edition, 1997.
23. Microsoft .NET web page. http://www.microsoft.com/net/.
24. S.L. Peyton-Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Proc. 23rd ACM Symposium on Principles of Programming Languages*, pages 295–308, 1996.
25. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.
26. Z. Somogyi, F. Henderson, and T. Conway. Mercury – an efficient, purely declarative logic programming language. In *Proc. Australian Computer Science Comference*, pages 499–512, 1995.
27. J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, second edition, 1992.
28. Sun Microsystems. Enterprise JavaBeans Specification. Version 2.0, 2001.
29. C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.