Collaboration and Composition: Issues for a Second Generation Process Language.

B.C. Warboys[1], D. Balasubramaniam[2], R.M. Greenwood[1], G.N.C. Kirby[2], K. Mayes[1], R. Morrison[2], D.S. Munro[3].

Department of Computer Science, University of Manchester[1]

School of Mathematical and Computational Sciences, University of St Andrews[2]

Department of Computer Science, University of Adelaide[3]

Abstract. Over the past decade a variety of process languages have been defined and applied to software engineering environments. The idea of using a process language to encode a software process as a "process model", and enacting this using a process-sensitive environment is now well established. Many prototype process-sensitive environments have been developed; but their use in earnest has been limited. We are designing a second generation process language which is a significant departure from current conventional thinking. Firstly a process is viewed as a set of mediated collaborations rather than a set of partially ordered activities. Secondly emphasis is given to how process models are developed, used, and enhanced over a potentially long lifetime. In particular the issue of composing both new and existing model fragments is central to our development approach. This paper outlines these features, and gives the motivations behind them. It also presents a view of process support for software engineering drawing on our decade of experience in exploiting a "first generation" process language, and our experience in designing and exploiting programming languages.

Keywords: process languages, process-based environments, persistence, concurrency control, hyper-programming

---

1. Informatics Process Group (IPG), Department of Computer Science, University of Manchester, Oxford Road, Manchester, M13 9PL, UK. email{brian,markg,ken}@cs.man.ac.uk

2. School of Mathematical and Computational Sciences, University of St Andrews, North Haugh, St Andrews, Fife, KY16 9SS, UK. email {dharini,graham,ron}@dcs.st-and.ac.uk

3. Department of Computer Science, University of Adelaide, South Australia 5005, Australia. email dave@cs.adelaide.edu.au

In O. Nierstrasz and M. Lemoine, editors, *European Software Engineering Conference (ESEC'99),*pages 75–91, Toulouse, France, September 1999. Springer-Verlag Lecture Notes in Computer Science 1687

1 Introduction

There is a long association between process languages and efforts to provide computer support for software engineering. Initially there was considerable work which concentrated on providing integrated project support environments to support users working together. The emphasis was mostly on tool integration; ensuring that the design tools, compilers, debuggers etc. could work together. It became apparent that support environments could make a greater contribution if they had knowledge of the process in which their users were involved. This led to further research on process-based environments [1,5,7]. The recognition that software processes can themselves be described as software is attributed to Osterweil [19], and has led to the development of process programming as part of software engineering, and ongoing research into process-centred environments.

More recently the interest in understanding and designing business processes, in particular the vogue for business process re-engineering, has led to an expansion of interest in process languages outside the software area. In general these have been simple languages specialised for an application area. For example, many forms-based workflow systems have a language based on the notion of passing an electronic document around an organization [14]. The concentration has been on high-volume processes where standards are key to operational efficiency, e.g. car loan credit checking.

Over the past decade the Informatics Process Group (IPG) at Manchester has developed a number of process models using the first-generation language of ICL's ProcessWise Integrator (PWI) [20,24] and more recently ProcessWeb [9,29] which combines PWI with a Web interface. We believe that our experience has been typical. First-generation languages, and the systems which evaluate them to provide process systems, are technically feasible and promising [7]. However the costs of developing process models are too high. Too great a knowledge of the language implementation is needed to develop effective models, and the code, when completed, tends to give a somewhat obscure representation of the process flow. This is partly because many first-generation languages have adopted specific implementations, particularly with respect to modelling collaboration, which have restricted their range of applicability.

Research in the Persistent Programming Research Group (PPRG) at St Andrews has tackled the problems of constructing and maintaining large, long-lived application systems, including software development environments [16,18]. Many of these problems are closely related to process language issues and therefore the techniques developed can be refined and incorporated in a process language. The two techniques discussed in this paper are Communicating Actions Control System (CACS) [22] and hyper-programming [13,17]. CACS addresses the problem of providing flexible concurrency control mechanisms to support collaborative working. Hyper-programming provides a novel approach to developing long-lived systems through allowing new code to be not just text but both text and explicit links to existing code and data.

Together the IPG and PPRG are now developing a second-generation language based on a synthesis of our joint experiences [26]. We see process models as fulfilling a key role in modern computer systems; that of relating the business processes and the IT systems which support them [27]. Understanding this relationship is growing in im-

In O. Nierstrasz and M. Lemoine, editors, *European Software Engineering Conference (ESEC'99),* pages 75–91, Toulouse, France, September 1999. Springer-Verlag Lecture Notes in Computer Science 1687

portance as systems are increasingly knitted together from existing components, and must have the flexibility to adapt in response to business changes. Automation involves introducing new IT systems into existing systems and changing the business processes to exploit them. This has led us to place emphasis on how our second generation language represents collaboration, and how models can be developed by composing components.

In designing a process language we are addressing the problems which we have experienced with first generation languages. We also have a view of the kind of process system which we want to express in our new language which derives from our work on process modelling methods.

2 Motivation - A Process System

A number of process systems which execute programs in process languages have been developed. These systems have been strongly influenced by factors such as tool invocation, visualisation, and meta-processes which do not figure so highly in traditional program language design. There are a number of contributing factors:

- The contribution of people is part of the process. A process system can support the people involved by ensuring that the right information is in the right place, at the right time. However, people are biddable rather than controllable by the process system. For example, if input is requested there is no control over how long a user will take to respond.
- Processes can last for a long time. Software projects may last for weeks, months or even years. To support such processes, process systems must themselves last just as long.
- Process models are developed over a period of time. Early parts of a software development project often involve investigating alternatives and making decisions about the future course of the project. To support this a process model cannot be fully defined when its enactment starts. A process system must support the incremental development of models and provide facilities which bind new and existing model fragments.

The general scheme adopted by most process languages is to model the process as a production system. The software product is the output of the software development process. This process is usually described in terms of a set of partially ordered tasks (activities) with output to input connections between them. This leads to a factory view where there is a "master process program" which provides the instructions to keep everything running efficiently; the emphasis is on design for efficiency.

Our preference, based on experience in modelling processes both inside and outside the software domain, is to view the process system as a service system. The purpose of the process system is to provide effective assistance to its users. Software development requires the collaboration of many people over a period of time. A process system can provide information to people, ensure that they do not mistakenly work on incorrect, or incomplete, data, and it can carry out some routine, mundane activities on behalf of its users. From a systems theory perspective, the process system is a serving system which

In O. Nierstrasz and M. Lemoine, editors, *European Software Engineering Conference (ESEC'99),*pages 75–91, Toulouse, France, September 1999. Springer-Verlag Lecture Notes in Computer Science 1687

supports its users, a served system [3]. To be viable a process system must continue to support the changing requirements of its users; design for evolution is a key theme [25]. This implies a need to separate existing models into fragments which can be composed in new ways to address future requirements.

For a process system to support the collaboration requirements of its users, its process language must be able to express a rich range of collaborations. A process language needs to provide high-level concepts which offer flexibility in modelling and enable effective and efficient support systems to be developed.

3 Collaboration

Our experience, from prototype systems, industrial case studies, and developing a process modelling method, is that a focus on collaboration is an effective way of modelling processes [8,12,27]. Unfortunately whilst many first-generation languages offer good abstractions for activity, they offer only low-level communication mechanisms, without any real abstractions, with which to realistically model coordination [10].

In PWI's Process Management Language (PML) this collaboration is modelled in terms of explicit message passing using typed buffers called *interactions* [2,11]. Our second generation approach is to adopt a more abstract general view of collaboration as mediated access to shared data. The specific implementation of message passing in PWI's PML interactions can then be defined using this more abstract view. However many other forms of collaboration can also be defined.

3.1 A Small Example

Consider the case of a sub-process where there is one software engineer who writes or updates a module, and another who must check it. One implementation of this involves a "module writing" activity which results in a revised module delivered to a "module checking" activity. This latter activity either outputs a "checked module" for input to the next sub-process or delivers the module and comments back to "module writing". This implementation means that the module checker cannot start until the module writer has finished writing, and once the writer has submitted the module for checking, further module writing must wait until the checker finishes. In many cases, however, this is not how work progresses in reality: the module writer may deliver a draft version for checking and continue writing; a checker's comments may include updates to the module. What matters is that the two people involved have an agreed protocol for manipulating the module, and deciding when their tasks are complete.

Figure 1 provides a sketch of this small example. The overlap between "module writing" and "module checking" shows that these components collaborate. A collaboration, which is in essence shared behaviour, is defined by identifying the shared data involved and the rules for accessing it. Our approach to defining these sharing rules is outlined in the next section. A key part of this is the separation of the collaboration protocol from the details of a specific process. In this example there is a "writing-checking" collaboration protocol which is independent of the organizational rules on how modules should be written, and how they should be checked. We may want to reuse this protocol in the context of writing and checking design documents, test plans, user manuals etc.

In O. Nierstrasz and M. Lemoine, editors, *European Software Engineering Conference (ESEC'99),* pages 75–91, Toulouse, France, September 1999. Springer-Verlag Lecture Notes in Computer Science 1687

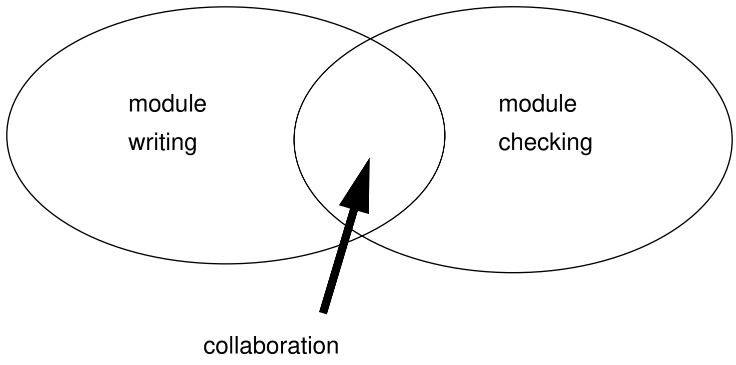

Figure 1 A small process showing the collaboration between two components

Similarly, we may want to change the details of how a module is checked without making changes to the collaboration protocol. This separation is one example of the kind of structuring which needs to be applied to process models in order to maintain their clarity.

We have written a series of related writer-checker processes in PWI's PML [2,11], Java, and Little-JIL [21,28]. In these implementations, the collaboration protocol was a significant proportion of the effort, and it became closely interwoven with the rest of the process. This restricted our ability to reuse fragments between related processes; our library of re-usable process components just did not grow as we had envisaged.

3.2 Communicating Actions Control System (CACS)

Communicating Actions Control System (CACS) is an abstract operational model designed to allow the specification of coherency protocols for accessing shared data [22]. The CACS model consists of actions (computations) that access objects (shared data).

A particular coherency protocol, for example atomic (ACID) transactions, is defined by a set of *significant events* and a set of *rules*. The significant events specify the operations on shared data that need to be coordinated by the protocol. The rules specify the details of this coordination. For example, consider the atomic transaction protocol:

- The significant events are: *begin*, *commit*, *abort*, *read* and *write*.
- The rules give an operational specification of how the ACID transaction properties are to be enforced.

As it runs, each action generates a sequence of significant events, which are handled by the CACS *controller*, according to the rules. Each rule specifies what the controller should do in response to a particular significant event. In addition to performing arbitrary computation, the controller may suspend and resume actions, and generate additional, synthetic events that are added to the incoming event stream.

In O. Nierstrasz and M. Lemoine, editors, *European Software Engineering Conference (ESEC'99)*, pages 75–91, Toulouse, France, September 1999. Springer-Verlag Lecture Notes in Computer Science 1687

```
let writerule[write,%id,%v] =
    if there is an unread write of shared data item %v
    then suspend %id on write %v
    else
    begin
        update %v
        if there is a read %id2 suspended on write %v
        do unsuspend %id2
    end
```

Figure 2 Example CACS Rule - write is the CACS event, %id the CACS action identifier and %v is the shared data item manipulated

This architecture gives considerable flexibility both in defining new coherency protocols, and in varying the policy implementing a particular protocol. For example, both optimistic and pessimistic flavours of atomic transactions could be defined in a similar way. The significant events would be the same in each case, but the bodies of the rules would vary. For an optimistic scheme the controller would allow actions to read and write shared data without restriction, recording which data objects had been accessed. On a *commit* event, the controller would test for conflict with other transactions, and generate an *abort* event for each conflicting transaction. For pessimistic transactions the controller would suspend an action on the first attempted conflicting access to shared data.

In general it is not possible for the CACS system to deduce automatically the points in an action at which significant events are generated. The source program must thus be annotated to indicate these. In some special cases, however, this may be done automatically. For example with atomic transactions the system may deduce where *read* and *write* events occur, but not *begin*, *commit* or *abort*.

CACS specifications, in terms of events and rules, can be written for a wide variety of coherency protocols. These include traditional schemes such as atomic transactions and monitors, and more complex application-specific schemes. The programming involved in the correct implementation of these schemes can be defined and placed in a library for reuse. The writer of a CACS action, a process computation in our case, thus does not have to write CACS specifications in cases where standard coherency protocols are sufficient, but has the flexibility to define new schemes if required. For example, it is possible to implement CACS rules to give the particular message-passing semantics which are currently offered by interactions in PWI's PML.

In the writer-checker example, one useful coherency protocol is that offered by a single element buffer. A data item must be written before it is read. Once written, the "buffer" is full and the item must be read before it can be written again. In CACS terms, a read event may suspend because it must wait for the corresponding write event, and a write event may suspend until the value from the previous write has been read. There would be two CACS rules, one handling read events and one handling write events. Figure 2 illustrates pseudo-code for the write rule.

```
Checker
shared data - checksd, cuisd
let chModule := nullModule
let chComments := nullComments
while checkernotfinished() do
begin
    chModule := checksd.module
            ! CACS event -[read,my_aid,checksd.module]
    cuisd.module := chModule
            ! CACS event -[write,my_aid,cuisd.module]
    chComments := cuisd.comnts
            ! CACS event -[read,my_aid,cuisd.comnts]
    checksd.comnts := chComments
            ! CACS event -[write,my_aid,checksd.comnts]
end
```

Figure 3 Example code for Checker (including annotations identifying CACS events)

One of the benefits of the CACS approach is that it does not restrict the number of computations which access a specific shared data area. This means that collaborations involving more than two parties can be modelled more naturally than with a message passing style. This was a particularly unpleasant problem to implement using the message-passing semantics of PWI's PML interactions.

An example checker is given in Figure 3, corresponding to the checker action shown in Figure 4. As the shared data is explicitly identified, we can anticipate that the compiler will be able to generate the CACS events, here shown in comments. The shared data, *checksd*, is shared between checker and writer. It contains two fields, *module* and *comnts*. This checker simply loops through four steps. The first step is to read the module from the shared data, *checksd*, and assign it to checker's local variable, *chModule*. It is at this point that checker will suspend if the module is not available. The second step makes the module available to the checker user, and the third step reads the comments from the user when they are available. The fourth step is to write those comments to the shared data, *checksd*, and so make them available to the writer. The function *checkernotfinished* returns a boolean value which is **true** when the process is complete. If there is only one checker this is simply when the comments indicate the module is acceptable, but with multiple checkers this can be more complicated.

3.3 User and Tool Interaction as Collaboration

Figure 3 is code which will be executed by the process system to support the user responsible for the checking. It makes the module available to this user by writing to a shared data area. Figure 4 depicts an overview of the module example showing the two users. Over time the boundary between what is done by the system and what is done by the users may change. For example, if checking only involved ensuring that the module

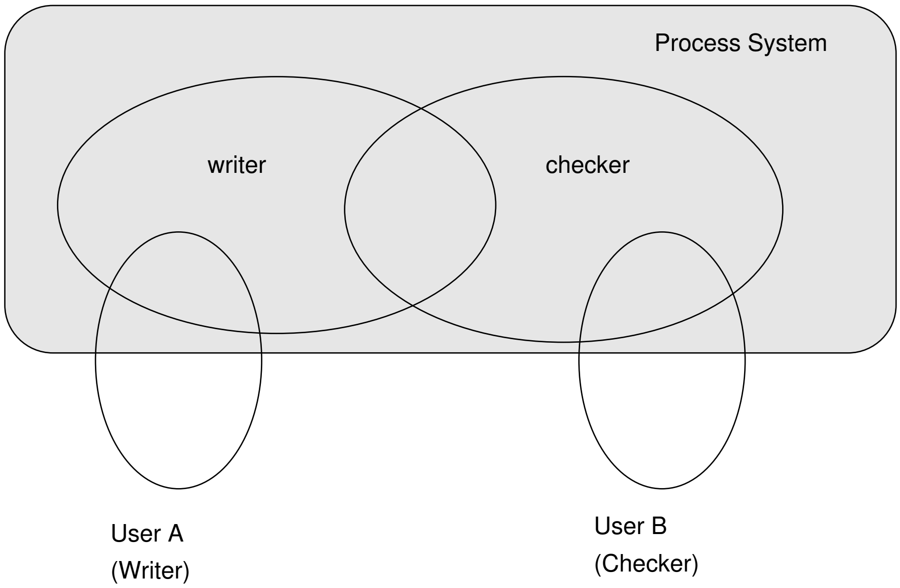

Figure 4 User Interaction as Collaboration

compiled and conformed to coding standards, then the organization might decide to invest in tools to automate this. In our process language we want to be able to model such automation. This gives us a requirement that collaboration with the external world, both users and external tools, should be handled in the same way as collaboration within the process system. This is represented as the manipulation of data shared between a computation within the process system and an "external computation" carried out by a user or another software tool.

A process model in our system is represented by a persistent, strongly typed collection of data and programs held within the system, interacting with other software and users outside the system. The hyper-program, to be described in the next section, is a representation which provides a single consistent description format for everything within the system. Input/output then involves a translation between the strongly typed internal form and other external forms. For interaction with a user these external forms might be X-Windows events or HTTP streams; for interaction with other tools they might be raw bytes, database relations etc.

A particular sequence of Input/Output can itself be viewed as a collaboration between the outside world and an action within the process model that implements the required translation. CACS rules can thus be used to coordinate the Input/Output. Again, a library of pre-defined rule sets for standard Input/Output patterns can be provided.

In O. Nierstrasz and M. Lemoine, editors, *European Software Engineering Conference (ESEC'99),* pages 75–91, Toulouse, France, September 1999. Springer-Verlag Lecture Notes in Computer Science 1687

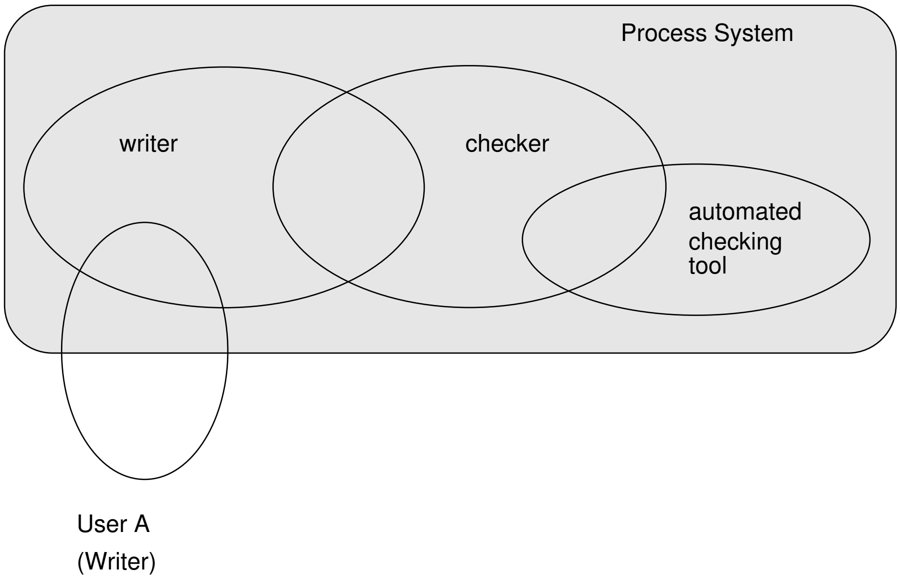

Figure 5 Automation through moving the process system boundary

4 Composition

A significant motive for our focus on collaboration was the kind of process systems which our experience tells us it is desirable to build. Similarly, the way we build such systems is a significant motive for our composition focus. In this context there is a need to address not only the issue of how components are composed, but also the facilities to decompose assemblies of components and re-compose them in new ways.

Our view is that a model developer should be thinking about composing fragments, rather than writing a model from scratch. This gives a close mapping between the way that a model is understood as a set of mediated collaborations, and the way that it is developed. One motivation for this is to encourage re-use of existing model fragments as a standard part of model development. A second motivation stems from the potential longevity of our process models.

Figure 5 depicts the automation of the checking in our example through replacing “User B” with a computation within our process system. If the “automated checking tool” uses the same shared data and protocol as “User B” there is no need for any changes to “checker”. If we had replaced “User B” with an external tool, the diagram would have been the same as Figure 4 with “User B” re-labelled “external checking tool”. Again our focus would have been on the collaboration between this tool and “checker”.

The examples in Figure 4 and Figure 5 show a close relationship between modelling based around collaboration and a modelling method in which composition plays a

In O. Nierstrasz and M. Lemoine, editors, *European Software Engineering Conference (ESEC’99),*pages 75–91, Toulouse, France, September 1999. Springer-Verlag Lecture Notes in Computer Science 1687

strong role. The intuition from the figures is that in both cases "writer" and "checker" will be composed in the same way. A typical concrete approach to sharing such as interactions, also defines, by implication, a composition approach. Our abstract sharing model does not. We need to have composition facilities to enable us to construct specific sharing implementations in the language.

In the context of our small example, there are many possible enhancements to a model which supports a single writer and single checker. In some situations it may be better to have two checkers who must both agree that the module is acceptable. This could be made visible to the writer, or it could be hidden so that the writer collaborates with one, two, or more checkers in exactly the same fashion. With more than two checkers there might be some kind of voting algorithm to determine if the module was acceptable. Writing and checking might form part of a larger quality control process, such as Fagan inspections. Our goal in using CACS is to allow the programmer to carry out these kinds of adaptations as easily as possible.

4.1 Hyper-Programming for Model Development

The technique which we adopt is to base our model development on hyper-programming [13,17]. Traditionally, a program which accesses another potentially shared object during its execution, contains a textual description of how to locate the object. This description is subsequently resolved, commonly during linking for code objects and during execution for data objects. This resolution is necessary because programs are constructed and stored in some long-term storage facility, such as a file system, which is separate from the run-time environment which disappears at the end of each program's execution. By contrast in a persistent programming system, programs may be constructed and stored in the same environment as that in which they will subsequently be executed. This means that objects accessed by a program may already exist when the program is composed. Direct links to the objects can be included in the program rather than textual descriptions of the access paths by which they can be located at execution time. A program containing both text and links to objects is a hyper-program.

There are a number of benefits of hyper-programming [13].

- early checking - access path checking and type checking for linked components can be performed during program construction,
- associations between executable programs and their source programs can be maintained automatically,
- source representations of all programs, including those that may, due to the context in which they were defined, contain references to other existing data - it is difficult to fully describe such programs with a purely textual notation.

These benefits are all relevant in our context of a process language and system which supports model development through composition and incremental evolution. The ability to include explicit links to existing objects closely matches development through composition. Part of a modeller's task is to describe how existing fragments (objects) should be assembled to produce the new model required. The ability to relate executable and source descriptions means that process model instances and their source descriptions can be related over their lifetime rather than just at instantiation time. The ability

In O. Nierstrasz and M. Lemoine, editors, *European Software Engineering Conference (ESEC'99),* pages 75–91, Toulouse, France, September 1999. Springer-Verlag Lecture Notes in Computer Science 1687

to have hyper-program representations for both source and run-time makes incremental definition of long-lived models significantly simpler.

4.2 Returning to our Small Example

Hyper-programming is very helpful in enabling a development by composition approach. We can develop the components separately and then use hyper-links to make the appropriate connections. We can also unlink and re-link as required. If we consider the *writer-checker* example there are several possibilities for components being developed at different times. CACS specifications for common coherency protocols are available in a library when the *writer-checker* model is being developed. Hyper-programming enables the use of these in *writer-checker* to be checked at compile time. Simple mistakes can be eliminated when the model is being developed rather than only becoming evident at run-time. Another process might want to use the checked module once *writer-checker* is complete. A compositional development approach, supported by hyper-program links to existing fragments (objects), is well suited to developing models and libraries over a period of time.

In most first generation process languages the traditional development sequence is: write a process definition; specialise the definition into an enactable process; and spark the enactable process to yield an enacting process [6]. This means that a single process definition can be specialised and sparked many times to give separate enacting processes. The distinction between the definition (source) and enacting (run-time) representations makes it more difficult to compose new definitions with fragments of existing enacting processes. In progressing from the definitions of the separate components in *writer-checker* to an enacting model, which is supporting two users writing and checking a particular module, we make use of the ability of hyper-programs to provide both source and run-time representations. When writer and checker are developed they are defined as *scripts*. A script corresponds to a CACS action. It records business rules in terms of the sequencing of operations which manipulate local script data and mediated shared data. (For example Checker in Figure 3.) A script is a piece of code and suspended thread. When defined it is suspended at the start of its execution. The hyper-program which is compiled to produce a script is also the hyper-program which represents the script in its initial suspended state. The system has a built in function *activate* which is used to spark a collection of scripts, returning an *activity*. (An activity thus corresponds to a set of CACS actions, along with associated shared data and rules.)

Figure 6 illustrates activate sparking the scripts in the *writer-checker* example. The activate takes two parameters: a set of scripts and a set of rules. Thus the *activate* would be passed the Checker script code from Figure 3 and the writerule from Figure 2. The scripts mention the CACS events, while the rules provide the implementation of the events in a particular concurrency protocol. For example, in a case where the scripts used a transaction protocol there might be one set of rules which implemented optimistic locking and another set which implemented pessimistic locking. If we want to have multiple instances of *writer-checker* then we can write a constructor function which returns an activity, and call it as often as required.

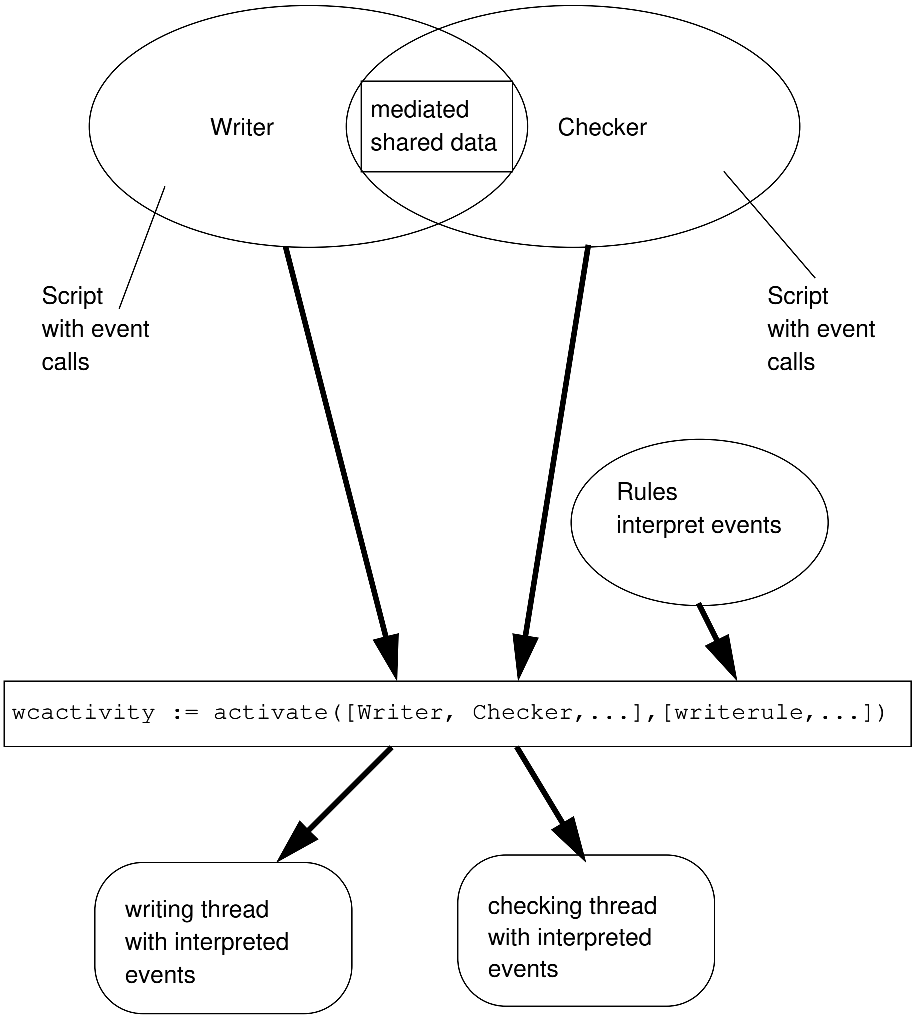

Figure 6 Using activate to convert scripts into an enacting process

In O. Nierstrasz and M. Lemoine, editors, *European Software Engineering Conference (ESEC'99),* pages 75–91, Toulouse, France, September 1999. Springer-Verlag Lecture Notes in Computer Science 1687

```
! suspend activity and extract a script
suspendedwc := decompose ( wcactivity );
suspendedChecker :=
        suspendedwc.scriptvector[checkerIndex];

! revise script and replace in suspended activity
revisedChecker := changefunction( suspendedChecker );
suspendedwc.scriptvector[checkerIndex] := revisedChecker;

! restart the suspended activity
wcactivity := activate( suspendedwc );
```

Figure 7 An example of stopping, modifying and restarting an activity using decompose

There is also a *decompose* function which takes an activity and returns the scripts in a suspended state. An activate following a decompose restarts the scripts from their suspension point as if the decompose had never occurred. It is also possible to use decompose and activate to dynamically compose or change enacting models. It may be that after decomposition some new scripts are added to the collection before it is activated, or individual scripts might be replaced, as shown in Figure 7.

5 Related Work

There are different schools of thought about the appropriate current research goals in the area of process languages. There are those who see new languages as an irrelevance which create artificial barriers between the research community and industrial practitioners. They place emphasis on exploiting and inter-operating with existing tools such as configuration management systems, workflow systems, object request brokers etc. [4]. This work is important but this does not mean that there is no need for further research on process languages. It has been noted that despite the considerable research in process languages, almost no novel approaches to software development have emerged. This suggests that the first-generation languages have been overly constrained by the emphasis on describing, promoting and supporting existing software processes. It also supports our thesis that many of these languages made an early commitment to particular mechanisms, which both make them difficult for inexpert modellers to use, and make it clumsy, or impossible, to represent some forms of collaboration.

The "second generation" school advocate that the lessons learned from first generation languages now need to be consolidated and exploited through the development of new process languages. Here the chief difference seems to be between those who believe the most promising approach is a set of sub-languages which can be factored together as and when required [23], and those like ourselves who are concentrating on a better core language [15]. One common theme is the issue of managing concurrency.

In O. Nierstrasz and M. Lemoine, editors, *European Software Engineering Conference (ESEC'99),* pages 75–91, Toulouse, France, September 1999. Springer-Verlag Lecture Notes in Computer Science 1687

Little-JIL [21] is a sub-language of the second generation process language JIL [23] which concentrates on the coordination of activities and agents. It has a visual syntax and is aimed at making it easier for practitioners to experiment with process programs. This emphasis on coordination is closely related to our view of processes as sets of mediated collaborations. In both it is recognised that an important part of managing concurrency is expressing how shared resources are handled. Another common theme is the importance of well-defined semantics to enable reasoning about processes written in second generation languages. This is one area in which our process modelling method [27] needs further improvement.

6 Conclusions

Interest in understanding processes and how we can apply computer systems to support them continues to drive the development and use of process languages. In general modelling software engineering processes has turned out to be particularly challenging and has therefore often acted as the driver in process language design. However, process languages should not be evaluated just on how accurately they can reflect established best practice in software engineering, producing traditional applications in traditional ways. There is a need to support rich collaborative protocols between a process system and users, and between a process system and other software tools. Viewing a process as a set of mediated collaborations is clearly a very general approach, within which particular collaboration styles can be defined. A common approach to collaboration whether within the process system or across its boundaries supports a clean and general approach to modelling process automation.

As processes may be developed incrementally over a period of time, the models to support them must be incrementally developed too. The process language and system must provide facilities which combine new process model fragments and existing enacting models. A collaborative viewpoint matches well with an approach based on generic composition facilities. These can be used to recombine existing process model fragments, and further to retain the composition structure to assist future understanding and change.

In developing a second generation process language we have been motivated by the problems in using existing languages, and a recognition of the key role of process languages in modern flexible architectures. We need a better understanding of process languages in order to meet the next challenge of developing appropriate architectures for the support and integration of process systems. By exploiting CACS and hyper-programming we are able to solve the issues of collaboration and composition in the style shown. This gives us an abstract machine which addresses collaboration and composition at a basic level and can be used as the target for other higher-level process representations.[1]

1. This work is supported by UK EPSRC grants GR/L34433 and GR/L32699, Compliant System Architecture

Bibliography

[1] Ambriola, V., Conradi, C. and Fuggetta, A. "Assessing Process-Centered Software Engineering Environments". *ACM Transactions on Software Engineering and Methodology*, 6(3), pp 283–328, July 1997.

[2] Bruynooghe, R.F., Parker, J.M. and Rowles, J.S. "PSS: A System for Process Enactment", in Proceedings of the First International Conference on the Software Process, pp 142–158, Redondo Beach, California, USA, 1991.

[3] Checkland, P. and Holwell, S. "*Information, Systems and Information Systems: making sense of the field*". John Wiley and Sons Ltd. 1998.

[4] Conradi R. and Liu, C. "Process Modelling Languages: One or Many", in Schafer, W. (Ed) *Software Process Technology: Fourth European Workshop EWSPT'95*, pp 98–118, Noordwijkerhout, The Netherlands, Springer-Verlag LNCS 913, 1995.

[5] Derniame, J.-C., Kaba, B.A. and Wastell, D. (Eds) "*Software Process: Principles, Methodology, and Technology*". Springer-Verlag LNCS 1500, 1999.

[6] Feiler, P.H. and Humphrey, W.S. "Software Process Development and Enactment: Concepts and Definitions", in *Proceedings of the Second International Conference on the Software Process*. pp 28–40, Berlin, Germany. IEEE Computer Society Press, 1993.

[7] Finkelstein, A., Kramer, J. and Nuseibeh, B, (Eds) "*Software Process Modelling and Technology*". Research Studies Press Ltd. 1994.

[8] Greenwood, R.M., Robertson, I., Snowdon, R.A., Warboys, B.C. "Active Models in Business", in *Proceedings of 5th Annual Conference on Business Information Technology (BIT'95)*, Department of Business Information Technology, Manchester Metropolitan University. 1995.

[9] Greenwood, R.M. and Warboys, B.C. "ProcessWeb - Process Support for the World Wide Web", in Montangero, C. (Ed) *Software Process Technology: Fifth European Workshop EWSPT'96*, pp 82–185, Nancy, France, Springer-Verlag LNCS 1149, 1996.

[10] Henderson, P and Pratten, G.D. "POSD - A notation for presenting complex systems of processes", *Proceedings of the First IEEE International Conference on Engineering Complex Computer Systems*, 1995.

[11] ICL. "ProcessWise Integrator PML Reference", ICL/PW/635/01, issued with Release 4.1, 1996.

[12] Kawalek, P., "*A Method for Designing the Software Support of Coordination"*, Ph.D. Thesis, University of Manchester, UK, 1996.

[13] Kirby, G.N.C., Connor, R.C.H., Cutts, Q.I., Dearle, A., Farkas, A.M. and Morrison, R. "Persistent Hyper-Programs". In *Persistent Object Systems*, Albano, A. & Morrison, R. (ed), Springer-Verlag, Proc. 5th International Workshop on Persistent Object Systems, San Miniato, Italy. In Series: Workshops in Computing, van Rijsbergen, C.J. (series ed) pp 86–106, 1992.

[14] Lawrence, P. (Ed) "*Workflow Handbook 1997*", John Wiley and Sons (in association with Workflow Management Coalition WfMC), 1997.

[15] Montangero, C. "In favour of a Coherent Process Coding Language", in Schafer, W. (Ed) *Software Process Technology: Fourth European Workshop EWSPT'95*, pp 94–97, Noordwijkerhout, The Netherlands, Springer-Verlag LNCS 913, 1995.

In O. Nierstrasz and M. Lemoine, editors, *European Software Engineering Conference (ESEC'99),* pages 75–91, Toulouse, France, September 1999. Springer-Verlag Lecture Notes in Computer Science 1687

[16] Morrison, R., Connor, R.C.H., Cutts, Q.I. and Kirby, G.N.C. "Persistent Possibilities for Software Environments". *In The Intersection between Databases and Software Engineering*, IEEE Computer Society Press, Proceedings ICSE-16 Workshop on the Intersection between Databases and Software Engineering, Sorrento, Italy, pp 78–87, 1994.

[17] Morrison, R., Connor, R.C.H., Cutts, Q.I., Dunstan, V.S. and Kirby, G.N.C. "Exploiting Persistent Linkage in Software Engineering Environments". *Computer Journal* 38 (1) pp 1–16, 1995.

[18] Morrison, R., Dearle, A., Bailey, P.J., Brown, A.L. and Atkinson, M.P. "The Persistent Store as an Enabling Technology for Integrated Project Support Environments". In *Proceedings of the Eighth International Conference in Software Engineering,* pp 166–172, London, UK. IEEE Computer Society Press. 1985.

[19] Osterweil, L.J. "Software Processes are Software Too", in *Proceedings of the Ninth International Conference on Software Engineering*, pp 2–14, Monterey, California, USA. IEEE Computer Society Press. 1987.

[20] Snowdon, R.A. "An Introduction to the IPSE 2.5 Project". *ICL Technical Journal* 6 (3) pp 467–478, 1989.

[21] Staudt Lerner, B., Osterweil, L.J., Sutton Jr., S.M., and Wise, A. "Programming Process Coordination in Little-JIL", in Gruhn, V. (Ed) *Software Process Technology: Sixth European Workshop EWSPT'98*, pp 127–131, Weybridge, UK, Springer-Verlag LNCS 1487, 1998.

[22] Stemple, D. and Morrison, R. "Specifying Flexible Concurrency Control Schemes: An Abstract Operational Approach". in *Proceedings of 15th Australian Computer Science Conference*, pages 873–891, Hobart, Tasmania, 1992.

[23] Sutton Jr., S.M. and Osterweil, L.J. "The Design of a Next-Generation Process Language" in *Proceedings of the Joint Sixth European Software Engineering Conference and the Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp 142–158, Zurich, Springer-Verlag LNCS 1301, 1997.

[24] Warboys, B.C. "The IPSE 2.5 Project: Process Modelling as the basis for a Support Environment", in *Proceedings of the First International Conference on Software Development, Environments and Factories*, Berlin. Pitman Publishing, 1989.

[25] Warboys, B.C., "The Software Paradigm", *ICL Technical Journal*, 10 (1) May 1995.

[26] Warboys, B.C., Balasubramaniam, D., Greenwood, R.M., Kirby, G.N.C., Mayes, K., Morrison, R. and Munro, D. "Instances and Connectors: Issues for a Second Generation Process Language", in Gruhn, V. (Ed) *Software Process Technology: Sixth European Workshop EWSPT'98*, pp 137–142, Weybridge, UK, Springer-Verlag LNCS 1487, 1998.

[27] Warboys, B.C., Kawalek, P., Robertson, I. and Greenwood, R.M. "*Business Information Systems: a Process Approach*". McGraw-Hill. 1999.

[28] Wise, A. "Little-JIL 1.0 Language Report". Technical Report 98-24, Department of Computer Science, University of Massachusetts at Amherst, April, 1998.

[29] Yeomans, B.S., "*A Process-Based Environment for the Evolutionary Development of Large Software Systems"*, MSc. Thesis, University of Manchester, UK, 1997.

In O. Nierstrasz and M. Lemoine, editors, *European Software Engineering Conference (ESEC'99),* pages 75–91, Toulouse, France, September 1999. Springer-Verlag Lecture Notes in Computer Science 1687