Process Evolution in the Marvel Environment

Gail E. Kaiser Israel Z. Ben-Shaul

Columbia University, Department of Computer Science, New York, NY 10027

The primary goal of a process evolution tool is to guarantee that the pre-existing objectbase is consistent with respect to the newly installed process. We had previously proposed an approach that rejected changes to the process that might potentially introduce inconsistency [2]. We have more recently developed a much more powerful approach that accepts any new process model (which is syntactically correct and the corresponding schema evolution is possible), and automatically updates the objectbase according to the new process. This second approach was implemented in the Evolver tool for the MARVEL 3.x rule-based environment [4].

The key insight that makes our new approach tractable is that it is unnecessary for Evolver to analyze the contents of the objectbase to determine whether or not it will be consistent. Instead, Evolver compares the old and new process models, and determines the set of rules affected by changes related to consistency. Evolver then generates a list of all possible instantiations of the affected rules, considering only the types but not the contents of the objects in the objectbase. Then these rules are executed by MARVEL's process engine as if they had been normal user commands.

This approach relies on our distinction between consistency and automation in the process model [3]. In essence, when one rule would forward chain to another rule to maintain consistency, the second rule is considered an implication of the first, and by definition must be fired whenever the first rule is fired. If it is not possible to execute an entire chain defined recursively by such implications, that chain must be rolled back (i.e., the entire consistency chain executes as a transaction), and thus the opportunity for backward consistency chaining can never arise. In contrast, if one rule would forward or backward chain to another rule solely for automation purposes, the chaining is considered optional. Automation chaining may be explicitly turned off, if desired, or can be terminated at any rule boundary without rollback of the entire automation chain.

Evolver generates a graph reflecting only the consistency implications among rules in the new process model, and compares it to the consistency rule graph representing the existing process. Matching a rule with its replacement may require interaction with the process engineer, since MARVEL allows multiple rules with the same name but different conditions. Evolver detects cases where consistency is either strengthened (adding an edge) or weakened (deleting an edge). The rule at the tail of each such edge must be evaluated, in the former case to consider rules that have become part of a previously executed consistency chain and in the latter case to consider rules whose conditions might have become satisfiable.

Evolver generates a batch script of MARVEL commands to fire any consistency (sub)chains necessary to make the objectbase consistent with respect to the new consistency graph. The script is executed in the MARVEL command line client, an alternative to the graphical user interface. Rule changes that are concerned only with automation, and do not affect consistency, are also installed but do not cause any updates to the objectbase. In addition to process evolution, Evolver also supports schema evolution based on facilities developed for Orion [1].

We now give a small example of actual evolution to the C/Marvel environment that we use in our own software development. C/Marvel consists of a data model (schema), a process model and a set of tool envelopes, based on our original organization of the MARVEL code in the file system, our manual development process and the corresponding Unix utilities, respectively. We used the Marvelizer immigration tool [7] to construct a C/Marvel objectbase containing the source, headers, libraries, executables, documentation, etc. of MARVEL itself.

C/Marvel's data model divides the environment's objectbase into a shared repository, called the "master area", and a collection of private workspaces, each called a "miniproject". As part of the process, a user initiates a code change by first reserving relevant objects in the master area and then copying them to a miniproject. The user does all editing and testing in the miniproject. Archived libraries in the master area may be linked together with modified code for testing, if objectbase links have been established from objects in the miniproject to the appropriate objects in the master area. Once the changes and test results are satisfactory, the user copies the objects back to the master area and deposits them. Whenever an object representing a source or header file is deposited, the affected archives and executables in the master area are marked as outdated in a mandatory consistency chain, and may be rebuilt then by an optional automation chain or later according to an explicit user command.

In our example, this process is modified so that outdated or reconstructed archives in the master area also propagate to all the miniprojects that link to them. The goal is to encourage users to incorporate changes in other parts of the system as soon as they have been deposited, rather than continuing testing using the old versions. We do not intend to claim the new process is necessarily better than the original process for any or all software development projects, just that it is representative of relatively simple but realistic process evolution.

```
arch [?a:AFILE]:
(and
 (exists LIB     ?l suchthat (member [?l.afiles ?a]))
 (exists PROJECT ?p suchthat (member [?p.lib    ?l]))
 (forall MODULE  ?m suchthat (linkto [?m.afiles ?a]))):
(and
            (?m.archive_status = Archived)
   no_chain (?p.status         = CompileAll))
  { }
  (?a.archive_status = Archived);

build [?mp:MINIPROJECT]:
 (and
   (forall CFILE ?c  suchthat (member [?mp.files ?c ]))
   (forall YFILE ?y  suchthat (member [?mp.files ?y ]))
   (forall LFILE ?l  suchthat (member [?mp.files ?l ]))
   (exists EXE   ?e  suchthat (member [?mp.exec  ?e ]))
   (forall AFILE ?a  suchthat (linkto [?mp.afiles ?a]))
   (forall MACHINE_EXEC ?me
                 suchthat (member [?e.machines ?me]))
   (forall MACHINE ?mc
                 suchthat (member [?a.machines ?mc]))):
 (and
   (?c.status = Compiled)
   (?l.status = Compiled)
   (?y.status = Compiled))
    { LOCAL build_local
         ?c.object_code ?l.object_code ?y.object_code
         ?mc.afile ?me.exec ?e.history "NO"}
   (?mp.build_status = Built);
   (?mp.build_status = NotBuilt);
```

Figure 1: Old C/Marvel arch and build Rules

Two of the old C/Marvel rules, arch and build, are shown in Figure 1. The parameter to arch is an AFILE object, which represents a Unix archive (".a") file. The arch rule first retrieves the ancestor PROJECT object containing this AFILE and all the MODULE objects linked to this AFILE. It then checks that all of the MODULEs linked to the AFILE have been Archived, and also that the containing PROJECT is in the CompileAll state. A PROJECT would be in this state if either a source file or header file had been deposited into the master area since the last time the PROJECT had been built (i.e., all its executables manufactured), and the PROJECT had not yet been rebuilt. This portion is there to prevent unnecessary work. If the condition already evaluates to true, or can be satisfied through backward chaining to other rules that archive all the relevant MODULEs, then the AFILE itself is said to have been Archived. No tool invocation is actually needed, since the condition directly implies the effect.

The parameter to the build rule is a MINIPROJECT object, which represents a private workspace for an individual software developer. The conditions gathers up all the C, yacc and lex source files contained in the MINIPROJECT, the MINIPROJECT's executable, and any AFILEs to which it is linked — perhaps in the master area or another miniproject. The MACHINE_EXEC and MACHINE objects are used to maintain archives and executables for different machine architectures. The build rule requires that all the source files have been Compiled since the last time the MINIPROJECT was built. If so, the activity part of the rule invokes the build_local envelope to rebuild the MINIPROJECT, that is, generate the executable being developed by the user. When the envelope terminates, one of the rule's two effects is asserted. If the build was successful, the MINIPROJECT is said to be Built; if the tool produces errors, however, it is NotBuilt.

Figure 2 gives modified arch and build rules, together with newly added update and restore rules. The only difference in arch is to change its one effect predicate from automation ("(...)") to consistency ("[...]"). When this effect predicate is asserted, consistency forward chaining must be attempted to every rule with a matching predicate in its condition. The new update rule has such a predicate.

The parameter of the update rule is a MINIPROJECT. When an archive in the master area causes arch to forward chain to update, the binding is ‘‘inverted’’ to find all MINIPROJECT objects that are linked to the particular AFILE object on which the consistency effect was asserted [6]. The update rule is then instantiated separately for each such object, and the condition is evaluated. If the MINIPROJECT is either INC_NotBuilt or NotBuilt, it is set to NotBuilt. Basically, NotBuilt means that the MINIPROJECT is out of date because one or more of the archives it imports from the master area has been updated since it was last built. INC_NotBuilt means that one of these archives

```
arch [?a:AFILE]:
 (and
  (exists LIB     ?l suchthat (member [?l.afiles ?a]))
  (exists PROJECT ?p suchthat (member [?p.lib ?l]))
  (forall MODULE ?m suchthat (linkto [?m.afiles ?a]))):
 (and
   (?m.archive_status = Archived)
    no_chain (?p.status = CompileAll))
   { }
   [?a.archive_status = Archived];

build [?mp:MINIPROJECT]:
 (and
  (forall CFILE ?c suchthat (member [?mp.files ?c ]))
  (forall YFILE ?y suchthat (member [?mp.files ?y ]))
  (forall LFILE ?l suchthat (member [?mp.files ?l ]))
  (exists EXE   ?e suchthat (member [?mp.exec  ?e ]))
  (forall AFILE ?a suchthat (linkto [?mp.afiles ?a ]))
  (forall MACHINE_EXEC ?me
              suchthat (member [?e.machines ?me ]))
  (forall MACHINE      ?mc
              suchthat (member [?a.machines ?mc ]))):
 (or
   (and
     (?c.status = Compiled)
     (?l.status = Compiled)
     (?y.status = Compiled)
     no_forward (?a.archive_status = Archived))
     (?mp.build_status = NotBuilt))
    { LOCAL build_local
         ?c.object_code ?l.object_code ?y.object_code
         ?mc.afile ?me.exec ?e.history "NO"}
    (?mp.build_status = Built);
    no_chain (?mp.build_status = NotBuilt);

update[?mp:MINIPROJECT]:
 (forall AFILE ?a suchthat (linkto [?mp.afiles ?a])):

 (and (?a.archive_status = Archived)
      (or no_backward (?mp.build_status = INC_NotBuilt)
          no_chain    (?mp.build_status = NotBuilt)))
    { }
    (?mp.build_status = NotBuilt);

restore[?mp:MINIPROJECT]:
 (forall AFILE ?a suchthat (linkto [?mp.afiles ?a])):
  no_backward (?a.archive_status = NotArchived)
  { }
  no_chain (?mp.build_status = INC_NotBuilt);
```

Figure 2: Modified and Added C/Marvel Rules

has been outdated but not yet reconstructed; this is set by the new `restore` rule when an imported archive becomes `NotArchived`. The effective difference is that a `MINIPROJECT` can be directly rebuilt when its status is `NotBuilt`, but it is first necessary to reconstruct the relevant archives in the master area if its status is `INC_NotBuilt`.

The modified `build` rule considers the relationship between a `MINIPROJECT` and the `AFILE`s it is linked to. If a source file within the workspace is recompiled and the imported archives have already been constructed, then the `MINIPROJECT` can be built. Alternatively, if an imported archive has just been reconstructed, so the status has become `NotBuilt`, it is also appropriate to now rebuild the `MINIPROJECT`. The no_chain directive was added to the second effect to prevent a cycle: if the `build` is unsuccessful, it is futile to immediately try again.

Evolver first detects that there were no changes to the data model, so no schema evolution is required. It then attempts to match rules in the old and new process model pairwise, and requests help interactively when there is an ambiguity. It discovers the ‘‘change’’ to `restore`, which strengthens consistency because it is the destination of a consistency effect of another rule, `touchup` (not shown). Evolver also finds the change to `update`, which is the destination of the new consistency predicate in `arch`. It then generates a MARVEL command script to trigger the offending `restore` and `update` rules on all affected objects. This would be all objects that are instances of the `MINIPROJECT` class or any subclasses of `MINIPROJECT` (there are none). The script is executed in a MARVEL command line client spawned by Evolver. This concludes our example.

Evolver's main limitation concerns evolution of product data manipulated by MARVEL's ‘‘black box’’ activities [5]. Consistency chains are currently restricted so that a non-empty activity can appear only in the original rule of a chain, and all rules triggered by forward consistency chaining must have an empty activity. Thus the rules queued by Evolver update only process data, as completely specified by the effects of rules, and never access product data. This *forward* repair approach would require human intervention for each non-empty activity that was invoked, defeating the purpose of an automated process evolution tool.

We postulate a *backward* repair solution, to apply to the rules in automation chains as well as to those in generalized consistency chains. The gist is to revert affected process data to their (new) default values, without actually undoing any activities. Then our current algorithm would be applied to move the process state forward to the degree possible through newly satisfied rules with empty activities. We hope to develop this idea further if resources become available.

The current Evolver tool has been incorporated into P/Marvel, a MARVEL environment for developing and evolving data models, process models and tool envelopes. A P/Marvel objectbase contains the source and internal representation of a data model and a process model, and includes references to both testing and "real" objectbases whose environments are instantiated by these models. Like any MARVEL environment, P/Marvel maintains consistency among its target objects, in this case components of data and process models, and automates aspects of the process, in this case installation and testing of new or evolved data and process models. P/Marvel is part of MARVEL version 3.0.2, which has been in use internally since July 1992, and is planned for release as part of MARVEL 3.1 in early 1993.

John Hinsdale implemented an initial version of Evolver that supported schema but not process evolution. Will Marrero worked on the Evolver described here, and George Heineman also contributed substantially to this research. The Programming Systems Laboratory is supported by National Science Foundation grants CCR-9106368 and CCR-8858029, by grants and fellowships from AT&T, BNR, Bull, DEC, IBM, Paramax and SRA, by the New York State Center for Advanced Technology in Computers and Information Systems and by the NSF Engineering Research Center for Telecommunications Research.

References

[1] Jay Banerjee and Won Kim. Semantics and Implementation of Schema Evolution in Object-Oriented Databases. In *ACM SIGMOD Annual Conference on the Management of Data*, pages 311-322. San Francisco CA, May, 1987. Special issue of *SIGMOD Record*, 16(3), December 1987.

[2] Naser S. Barghouti and Gail E. Kaiser. Scaling Up Rule-Based Development Environments. *International Journal on Software Engineering & Knowledge Engineering* 2(1):59-78, March, 1992.

[3] Naser S. Barghouti. Supporting Cooperation in the MARVEL Process-Centered SDE. In Herbert Weber (editor), *5th ACM SIGSOFT Symposium on Software Development Environments*, pages 21-31. Tyson's Corner VA, December, 1992. Special issue of *Software Engineering Notes*, 17(5), December 1992.

[4] Israel Z. Ben-Shaul, Gail E. Kaiser and George T. Heineman. An Architecture for Multi-User Software Development Environments. In Herbert Weber (editor), *5th ACM SIGSOFT Symposium on Software Development Environments*, pages 149-158. Tyson's Corner VA, December, 1992. Special issue of *Software Engineering Notes*, 17(5), December 1992.

[5] Mark A. Gisi and Gail E. Kaiser. Extending A Tool Integration Language. In Mark Dowson (editor), *1st International Conference on the Software Process: Manufacturing Complex Systems*, pages 218-227. IEEE Computer Society Press, Redondo Beach CA, October, 1991.

[6] George T. Heineman, Gail E. Kaiser, Naser S. Barghouti and Israel Z. Ben-Shaul. Rule Chaining in MARVEL: Dynamic Binding of Parameters. *IEEE Expert* 7(6):26-32, December, 1992.

[7] Michael H. Sokolsky and Gail E. Kaiser. A Framework for Immigrating Existing Software into New Software Development Environments. *Software Engineering Journal* 6(6):435-453, November, 1991.