Proceeding of the 1st ICEENG conference, 24-26 March, 1998.

MILITARY TECHNICAL COLLEGE
CAIRO-EGYPT

ICEENG 98

FIRST INTERNATIONAL CONF. ON
ELECTRICAL ENGINEERING

Automation Support for Concurrent Software Engineering

Salah Badr*

ABSTRACT

This paper presents an evolution control system that provides automated assistance for the software evolution process in an uncertain environment where designer tasks and their properties are always changing.
We view an Evolution Control System (ECS) as the agent that keeps track of proposed, ongoing, and completed changes to a software system. It provides automated assistance to the software evolution manager to help him/her make the right decisions. It automatically propagates change consequences by constructing the set of possibly affected modules. It also coordinates change implementation activities within the design team in a way that supports team work and maintains system integrity, as well as adapting itself to the dynamic nature of the evolution process where new changes arrive randomly and ongoing modifications are themselves subject to change as more information becomes available.

A. INTRODUCTION

An ECS has two main functions. The first is to control and manage evolving software system components (version control and configuration management, VCCM) and the second is to control and coordinate evolution team interactions in a way that maximizes the concurrent assignment and meets management constraints such as deadlines and precedences (planning and scheduling software evolution tasks which we refer to as evolution steps).
This system provides the required algorithms for coordinating and executing the activities mentioned above as well as the algorithms for reaching and maintaining a feasible schedule, if one exists, that meets the deadline requirements, reduces/avoids rollbacks, and insures system integrity in an uncertain environment where the set of evolution steps and their properties are always changing.

B. PREVIOUS WORK

The main areas in software engineering relevant to ECS are software development/evolution, version control and configuration management, task planning and scheduling, and concurrency control.

1. FORMAL EVOLUTION MODELS

In [9], Luqi presents a graph model for software evolution that introduced the notion of evolution step as the activities of initiation analysis and implementation of one request for change in the system under evolution. The graph model represents the software system evolution history as an acyclic bipartite graph G = {C, S, I, O}. C nodes represent system components and S nodes represent evolution steps. The input edges I represent the relation between a step and the set of system components that have to be examined to produce output components that are consistent with the rest of the system. The output edges O represent the relation between an evolution step and the components it produces. The states of an evolution steps as well as the generation of substeps to propagate the change consequences are also defined. In this paper we extend this graph model to include other relations among system components ("part_of" and "used_by") and the "part_of" relationship between composite step and its substeps.
Unlike the original development cycle, the evolution activities (adaptive, corrective, and perfective maintenance) must take into consideration the existing system's requirements, decomposition, constraints, capabilities and performance. The effect of the changes must be propagated to preserve system consistency. In the mean time, concurrent changes must be coordinated to avoid rollbacks and wasting engineering effort. Evolution changes must be planned so that they meet the management constraints such as deadlines.

* Dr. Salah Badr is the chief of information systems branch, Armament Authority of the Egyptian Armed Forces.

in each path that has only I and O edges. This represents the evolution history view of the graph. The edges represent the "part_of" (between a sub-component of a composite component and the composite component) and "used_by" relations (defined between components to represent the situation where the semantics or implementation of one component A depends on another component B; B used_by A) between the software components of a given configuration ($C'E \subseteq C' \times C'$), the "part_of" relation between a substep of a composite step and the composite step ($SE \subseteq S \times S$), the input relation between the system components which must be examined to produce output components that are consistent with the rest of the system and the corresponding evolution steps($I \subseteq C' \times S$), and output relation between evolution steps and the components they produce ($O \subseteq S \times C'$). System components are immutable versions of software source objects that cannot be reconstructed automatically.

An "edge_type" attribute is used to distinguish between the two kinds of edges representing the relations "used_by" and "part_of" defined on the set of edges $C'E \subseteq C' \times C'$. The "used_by" relation can be used for automatic identification of inputs of proposed evolution steps and identification of the induced steps triggered by a proposed step.

The model distinguishes between the primary and secondary inputs of a step. The primary input concept can be formalized by introducing the attributes object_id, version_id and variation_id of each version. Variations represent alternative choices, which may correspond to different formulations of the requirements in the context of prototyping, or different kinds of system software (operating system, window manager, etc.) in the context of product releases. Each variation is a linearly ordered sequence of versions. An input to a step is primary if and only if it is the previous version of the same object and belongs to the same variation as the output of the step.

1. Version and Variation Numbering

As soon as the input base version of a step is bound, the system assigns the version and variation number of the output object for the step. The variations are assigned successive numbers beginning with 1 for the initial variation. Versions along each variation are assigned successive numbers starting with 1 at the root version of the initial variation. This means that the new version number is the base version number plus one, while the variation number has two possibilities: the first possibility is to keep the base version's variation number at the time the step is assigned. This occurs when the base version is the most recent version on its variation line at the time the step is assigned. The other possibility is to use the "next" variation number, which is the highest variation number plus one. This labeling function illustrated in Figure 1 is the same for both atomic or composite objects (the entire software system is represented as a composite object).

This labeling function allows a version to belong to more than one variation which is a necessary modification to [9] to simplify the process of tracing the development history of a version and to keep a logical and realistic development history.

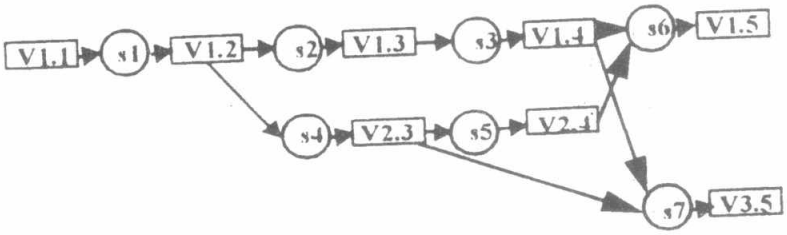

FIGURE 1. Variation and version numbering

2. States of Evolution Steps

The dynamics of the evolution steps are modeled by associating six different states with each step to express the different activities each step has to undergo during its lifetime. The state transition diagram in Figure 2 shows the different explicit decisions that have to be made by the management to cause the transition from one state to the other. It also shows the automated transitions from the scheduled state to the assigned state and vice versa (explained in detail in subsections c. and d below). By controlling the states of the evolution

a. Scheduling Tasks with Precedence Constraints

Scheduling tasks with arbitrary precedence constraints and unit computation time in multiprocessor systems is NP-hard for both the preemptive and nonpreemptive cases [12] [21]. Scheduling nonpreemptive tasks with arbitrary ready times is NP-hard in both multiprocessor and uniprocessor systems [12] [20] which excludes the possibility of the existence of a polynomial time algorithm for solving the problem. Hong and Leung [4] proved that there is no optimal on-line scheduler can exist for task systems that have two or more distinct deadlines when scheduled on m identical processors where m > 1.

Scheduling evolution steps to more than one designer with arbitrary precedence constraints and arbitrary deadlines is the same problem as that of multiprocessor scheduling mentioned above which is shown by many researchers to be NP-hard. These negative results dictate the need for heuristic approaches to solve scheduling problems in such systems.

In [16] Stankovic et al. present an O (n^2) heuristic scheduling algorithm for scheduling a set of independent processes on a set of identical processors. A task (process) in this model is characterized by an arrival time T_A, a deadline T_D, a worst case computation time T_C, and a set of resource requirements $\{T_R\}$. Tasks are independent, non periodic and non-preemptive. In [12], Ramamritham et al. introduce an O(nk) version of the algorithm introduced in [16] by considering only k tasks of the remaining tasks to be scheduled at each step. We have extended this algorithm to deal with precedence constraints and expertise levels of designers [1].

C. CONCEPTUAL MODEL

Since the main purpose of the ECS is managing software evolution in a rapidly evolving system, we review the graph model of software evolution that constitutes the context for building the ECS [9] [10]. The goal of this model is to provide a framework for integrating software evolution activities with configuration control [9]. The model of software evolution has two main elements: system components and evolution steps. System components are immutable versions of software source objects that cannot be reconstructed automatically. Evolution steps are changes to system components that have the following properties in the original version of the graph model [9]:

1. A top-level evolution step represents the activities of initiation, analysis, and implementation of one change request.
2. An evolution step may be either atomic or composite.
3. An atomic step produces at most one new version of a system component. This property is no longer true in our model in order to include the cases in which an atomic step is applied to an originally atomic component that needs to be decomposed according to some design considerations. This decomposition may lead to the production of more than one component. This modification is illustrated in section C.2.e later in this chapter.
4. The inputs and outputs of a composite step correspond to the inputs and outputs of its substeps.
5. The model allows steps that do not lead to the production of new configurations, e.g. design alternatives that were explored but not included in the configuration repository.
6. Completely automatic transformations are not considered to be steps and are not considered in this model.
7. The graph model can cover multiple systems which share components, alternative variations of a single system, and a series of configurations representing the evolution history of each alternative variation of a system.
8. A scope is associated with each evolution step which identifies the set of systems and variations to be affected by the step. The scope is used to determine which induced evolution steps are implied by a change request.

The evolution history is modeled as a graph G=[C, S, CE, SE, I, O]. This graph is a directed acyclic graph (bipartite with respect to the edges I and O). C and S are the two kinds of nodes (C: software component nodes, and S: evolution step nodes respectively). Each node has a unique identifier. C and S nodes alternate

Including the evolution steps, with all the data they have about the change they implement, as nodes in the bipartite evolution history graph facilitates evolution history tracing.
Our concept of composite entities and its generalization to fit system configurations is also similar to that used in PACT [13]. Our system uses a computed labeling function and a single versioning mechanism for automatically versioning individual objects as well as configuring a system (as a composite object). Simplifying version control and configuration management and making it transparent to the user without requiring his/her intervention, as it is the case in our system, are two of the main goals of a good version control and configuration management system as set forth by Feldman in [4].
According to Kaiser and Perry [4] the main tools that propagate changes among modules are listed below. However, none of these support the enforced model of cooperation among programmers necessary for large maintenance/evolution projects or automatically assign tasks to programmers:
Make: a UNIX tool that is used for regenerating up-to-date executables after source objects have been changed.
Build: is an extension to make that permit various users to have different views of target software system.
Cedar: the Cedar System Modeler uses an advanced version of the Make tool with version control to invoke the tools on a specific versions of files. This System informs the "Release Master", a programmer, about any syntactic interface errors. The Release Master is responsible for making work arrangements with responsible programmers.
DSEE: the Apollo Domain Software Engineering Environment also uses a Make-like tool with version control. DSEE also has a monitoring facility that permits programmers and/or managers to request to be notified when certain modules are changed.
Masterscope: Interlisp's Masterscope tool maintains cross-reference information between program units automatically. It also approximates change analysis of potential interference between changes by answering queries about syntactic dependencies among program units.
SVCE: the Gandalf System Version Control Environment performs incremental consistency checking across the modules in its database and notifies the programmer of errors as soon as they occur. The consistency checking is limited to syntactic interface errors. It supports multiple programmers working in sequence but does not handle simultaneous changes.
Kaiser and Perry [4] [7] [11] also describe Infuse, a system that automates change management by enforcing programmer cooperation to maintain consistency among a sequence of scheduled source code changes. Infuse automatically partitions these modules into a hierarchy of experimental databases but programmers are assigned to the these databases manually. This partitioning may be done according to the syntactic and/or semantic dependencies among the modules or according to project management decision. Consistency checking among the experimental database modules is a pre-condition for merging a database back to its parent experimental database (meaning that the interface between the modules must be correct and that the modules can compile and link successfully).
In our system tasks and copies of the associated versions of software components are assigned automatically to designers (programmers) according to their dependencies. Versions are generated automatically as soon as the work is done. Syntactic and semantic consistency checking for source code can be implemented by associating declarations of consistency constraints with steps, and triggering the required checking actions as part of the commit protocol.

3. APPROACHES TO SCHEDULING EVOLUTION STEPS

A scheduling problem in a real-time system is described by three basic concepts: the model of the system, the characteristics of the tasks to be scheduled, and the objective of the scheduling algorithm [12].
Task scheduling in real-time systems can be static or dynamic. A static approach performs the calculation of the schedules for tasks off-line. It requires prior knowledge of the characteristics of the tasks. On the other hand, a dynamic approach calculates schedules for tasks "on the fly". Static approaches have low run-time cost, but they are inflexible and cannot respond to a changing environment with unpredictable behavior. Dynamic approaches involve higher run-time costs, but they are flexible to adapt to environment changes. A survey of static and dynamic scheduling approaches can be found in [12].
Task scheduling can also be characterized as preemptive and nonpreemptive. A task is preemptive if its execution can be interrupted by other tasks and resumed afterwards. A task is nonpreemptive if it must run to completion once it starts.

precedence, and priorities. This indicates the need for an evolution control system that takes into account the special characteristics of the evolution (maintenance) phase of the software life cycle process that account for up to 75% of the cost of the software systems [15]

In the evolutionary prototyping model, where a prototype evolves via a number of versions to the final system, developers start evolving the software system from its fundamental concepts, then keep modifying the system in an interactive way with the customer until the system reflects the customer's real needs. The support provided by an evolution control system is particularly important in such an interactive, exploratory system development model because all kinds of changes are going on simultaneously, corrective changes to reflect the real customer requirements after reviewing the designer's interpretation of portions of the developed requirements, adaptive changes to the rest of the customer's real needs, and perfective changes to the fundamental concepts already accepted by the customers. The interactions between these different activities, the coordination among related ones, propagating the effects of each of these changes to the rest of the developed modules, and keeping track of which component belongs to which system version are the main goals of our evolution control system.

2. VERSION CONTROL AND CONFIGURATION MANAGEMENT

As indicated in [14], version control and configuration management is one of the fields in software engineering that has received much discussion and many proposals for proper version and configuration models in different domains, but little has been implemented, and much remains to be done in developing techniques for ensuring the consistency of configurations and space efficient algorithms for version management.

According to [8] and [3], representations of the versioning process can be classified into two main models. The first model is the conventional Version Oriented Model (VOM) in which a system is divided into modules each of which is versioned independently from the other modules. To configure a system one has to select a version of each module of the system. This makes *version* a primary concept while *change* is a secondary concept as a difference between versions. Both SCCS and RCS [17] [18] [19] conform to this model. The second model is the Change Oriented Model (COM). In this model the functional change is the primary concept. Versions are identified by a characteristic set of functional changes. To configure a system in this model, one has to select a set of mutually compatible functional changes. Versions in this model are global, meaning that to examine a module one has to specify a single version of the system first, then proceed to the required module. In a VOM system, to examine a module one has to select the module first, then individually select which version of this module is the target.

Reference [3] also defines the composition model and the long transaction model. The composition model is a natural outgrowth of the VOM model. A configuration in this model consists of a system model and version selection rules. A system model lists all the components of a system. Version selection rules define which version is to be selected for each component to compose a configuration rather than allowing the user to manually pick component versions.

The long transaction model supports the evolution of whole systems as a sequence of apparently atomic changes, and coordinates the change of software systems by teams of developers. Developers work primarily with configurations rather than individual components. A change is performed in a transaction. A specific configuration is selected as a starting point for changes which implicitly determines the version of the components. The modifications to this configuration are not visible outside the transaction until the transaction is committed. Multiple transactions are coordinated via concurrency control schemes to guarantee no loss of changes. The result of the committing of a transaction is a new system configuration version either on the same development path or branch from an existing development path resulting in a new alternative (variation) development path.

Our work utilizes concepts from both the VOM and long transaction models. Applying a top level evolution step to a base version of a software system leads to versioning of both the individual components involved in the change and the entire software system producing a new configuration version (version of a whole system). In addition our system automatically coordinates teamwork in such a way that concurrency control is done at a higher level of abstraction, i.e., the serialization of dependent evolution steps is done by serializing their assignment to developers in the same order and excluding the need for the traditional locking schemes

then the step status is automatically advanced to "assigned" and the designer is informed of the new assignment. When a step is assigned, the version bindings of its inputs are automatically changed from generic to specific. An edge is added as an input edge between the primary input component of the step and the step itself in the configuration graph.

e. Completed State

In this state the outputs of the step have been verified, integrated, and approved for release. This is the final state for each successfully completed step. This state can only be reached from the assigned state using the "commit_step" command. In this state the output components of the step have been added to the configuration graph. An output edge has also been added to the configuration graph between the step and its output component(s). A composite step enters the completed state when all of its substeps are completed

f. Abandoned State

In this state the step has been cancelled before it has been completed. The outputs of the step do not appear as components in the evolution history graph. All partial results of the step and the reasons why the step is abandoned are stored as attributes of the step for future reference. This is the final state for all steps that were not approved by the management or cancelled in the "approved", "scheduled" or "assigned" states.

3. SCHEDULING MODEL

The task in our case is to schedule a set of N evolution steps $S = \{S_1, S_2, ..., S_N\}$ relative to a set of M designers $D = \{D_1, D_2, ..., D_M\}$. The designers are of three possible expertise levels {Low, Medium, High}. Each step has associated with it a processing time tp (S_i), a deadline d (S_i), a priority p (S_i), and required expertise level e (S_i). Steps have precedence constraints given in the form of a directed acyclic graph G = (S, E) such that $(S_i, S_j) \in E$ implies that S_j cannot start until S_i has completed.

Because of the dynamics of the prototyping/evolution process, the steps to be scheduled are only partially known. Time required, the set of sub-tasks for each step, and the input/output constraints between steps are all uncertain, and are all subject to change as evolution steps are carried out.

Our goal is to dynamically determine whether a schedule (the time periods) for executing a set of evolution steps exists such that the timing, precedence, and resource constraints are satisfied, and to calculate this schedule if it exists.

D. DESIGN

The purpose of the Evolution Control System, ECS, is to provide automated support for changes in plan during the execution of the plan, and provide automatic decision support for planning and team coordination based on design dependencies captured in the configuration model. The ECS also manages the software evolution steps from its creation to completion and provides automatic version control and configuration management for the products of these steps.

a. Context Model

The Evolution Control System (ECS) interacts with two external entities: the software evolution manager and the software designer. These represent classes of human users rather than external software or hardware systems. There is one external interface for each class of user: the manager_interface and the designer_interface. Both of these interfaces are views of the proposed ECS. The message flow diagram in Figure 3 and the stimulus-response diagrams in Figures 7, 8, 9 and 10 show the context of the system and the available commands, their effects and the possible error conditions.

1. State Model And Related Concepts

The state of the ECS consists of a configuration graph, a schedule, a set of designers, and mappings giving the following attributes for each evolution step: deadline, estimated duration, precedence, priority, status and required expertise level. The formal definitions of the state model and the constraints on a feasible schedule are defined in [1].

steps, the evolution manager exercises direct control over both software evolution/development and the resulting software configurations. The following are the definitions of those states and the corresponding actions that cause the transition from one state to the other. These states are similar to those presented in [9] except that a new state called "assigned" has been added for the reasons explained below.

a. Proposed State

In this state a proposed evolution step is subjected to both cost and benefit analysis. This analysis also includes identifying the software objects comprising the input set of the step. A "proposed" step is generally added to the configuration graph as an isolated step node that does not have any input, output or part_of edges (except when an old version is used that has existing specific reference). This is because the primary and secondary input attributes are mostly generic inputs (object_id and variation_id only).

b. Approved State

In this state the implementation of the step has been approved but not scheduled yet and the input set of the step is not bound to particular versions. Approval of a proposed step by the management triggers the decomposition process to create an atomic sub-step for each primary or affected component of the step. These sub-steps inherit the status of their super-step which is "approved" in this case, and are added to the configuration graph with a part_of edge between each sub-step and its super-step. It is also in this state that the substeps are augmented with attributes that include the estimated duration of each sub-step and management scheduling constraints such as precedence, deadline, and priority.

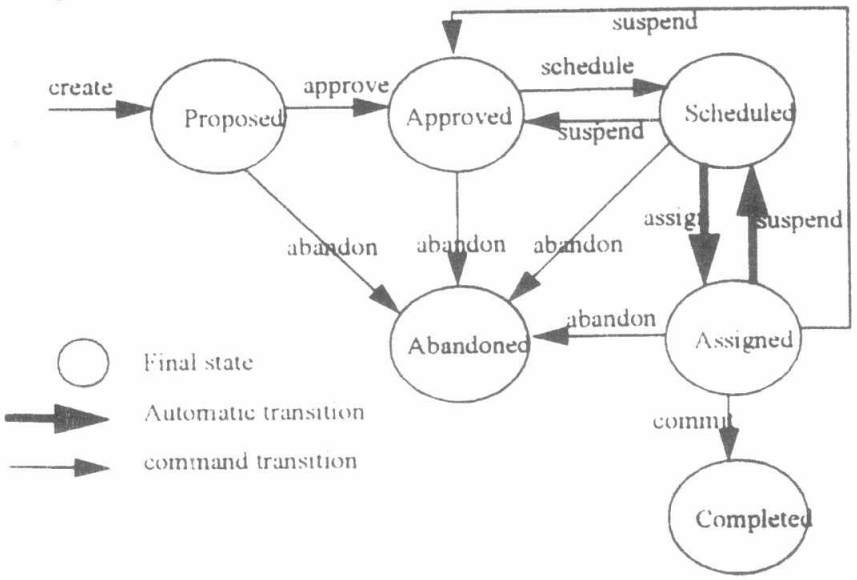

FIGURE 2. Evolution step's state transition diagram

*c. **Scheduled State***

In this state the implementation has been scheduled and the step is not yet assigned to a designer. The "scheduled" state is reached from the "approved" state via the command "schedule_step" that indicates that the management constraints are complete and enables the scheduling and job assignment mechanisms. The scheduling mechanism produces an updated schedule containing the newly scheduled step. A schedule specifies the expected starting and completion times for the step.

d. Assigned State

In this state the step is assigned to the scheduled designer, all inputs are bound to particular versions, and unique identifiers have been assigned to its output components, but these components are not yet part of the evolution history graph. A composite step enters the assigned state whenever any of its substeps is assigned. The assigned state is reached automatically from the scheduled state. When a designer is available, the schedule is used to determine his/her next assignment. If his/her next assignment is ready to be carried out

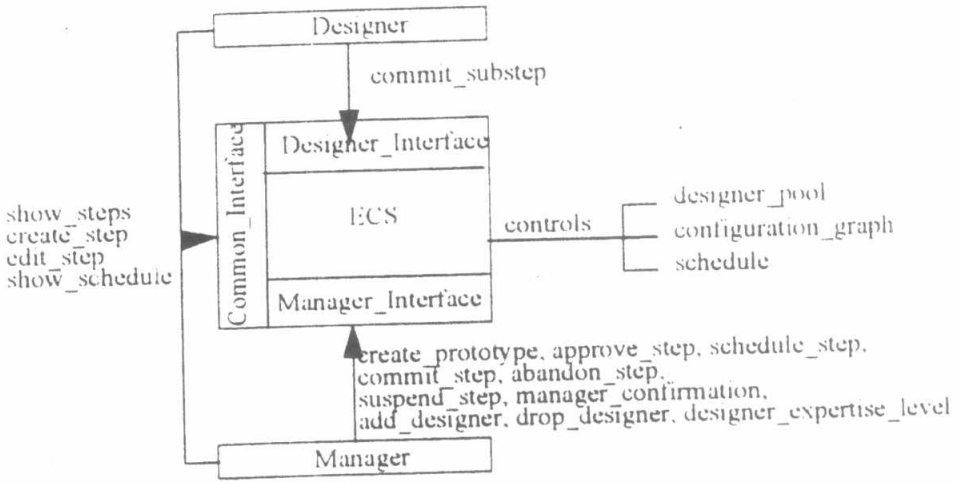

FIGURE 3. ECS message flow diagram

2. Interfaces

The manager interface to the ECS enables the manager to create new prototypes, provide for the evolution of the existing prototypes via a complete set of commands for creating, editing, scheduling, suspending/ abandoning and/or committing evolution steps, and manage the designer_pool data via add_designer, drop_designer, and designer_expertise_level commands. The designer interface to the ECS enables the designer to view the steps in a given prototype with a given status and get the sub-steps assigned to him. This

interface also enables the designer to create a sub-step of an assigned step as well as committing the assigned sub-step.The formal specifications of the various commands with the different responses for each command are defined in [1].

The following parameters can be adjusted manually (using the edit_interface) as uncertainties are resolved and planning errors are corrected. 1. Affected modules (Add/del). 2. Secondary input (Add/del). 3. Constraints (Precedence, Priority, Deadlines) (Initialize/Update). 4. Estimated duration (Update). 5. Resource (Designer Pool Changes) (Add/drop, Update).

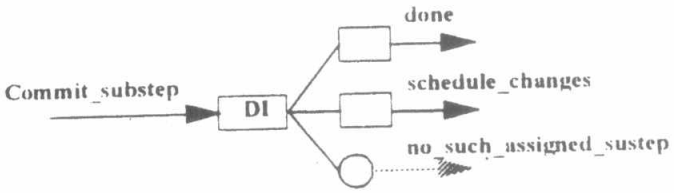

FIGURE 4. Stimulus-Response diagram for the designer interface

The schedule_step command triggers the scheduling mechanism that finds a feasible schedule if one exist or suggest changes to the deadlines of the lower priority steps until a feasible schedule is reached. When a designer is available for his assignment the ECS automatically checks out the required components from the design database to the designer's workspace and sends an e_mail message to the designer informing him about his new assignment. When a designer finishes his assignment, he simply issues the commit_step command. The system then automatically checks in the modified components to the design database giving them the right version and variation numbers and binding them to the appropriate configuration.

The ECS automatically monitors changes in plan and takes the appropriate action to maintain the required constraints.

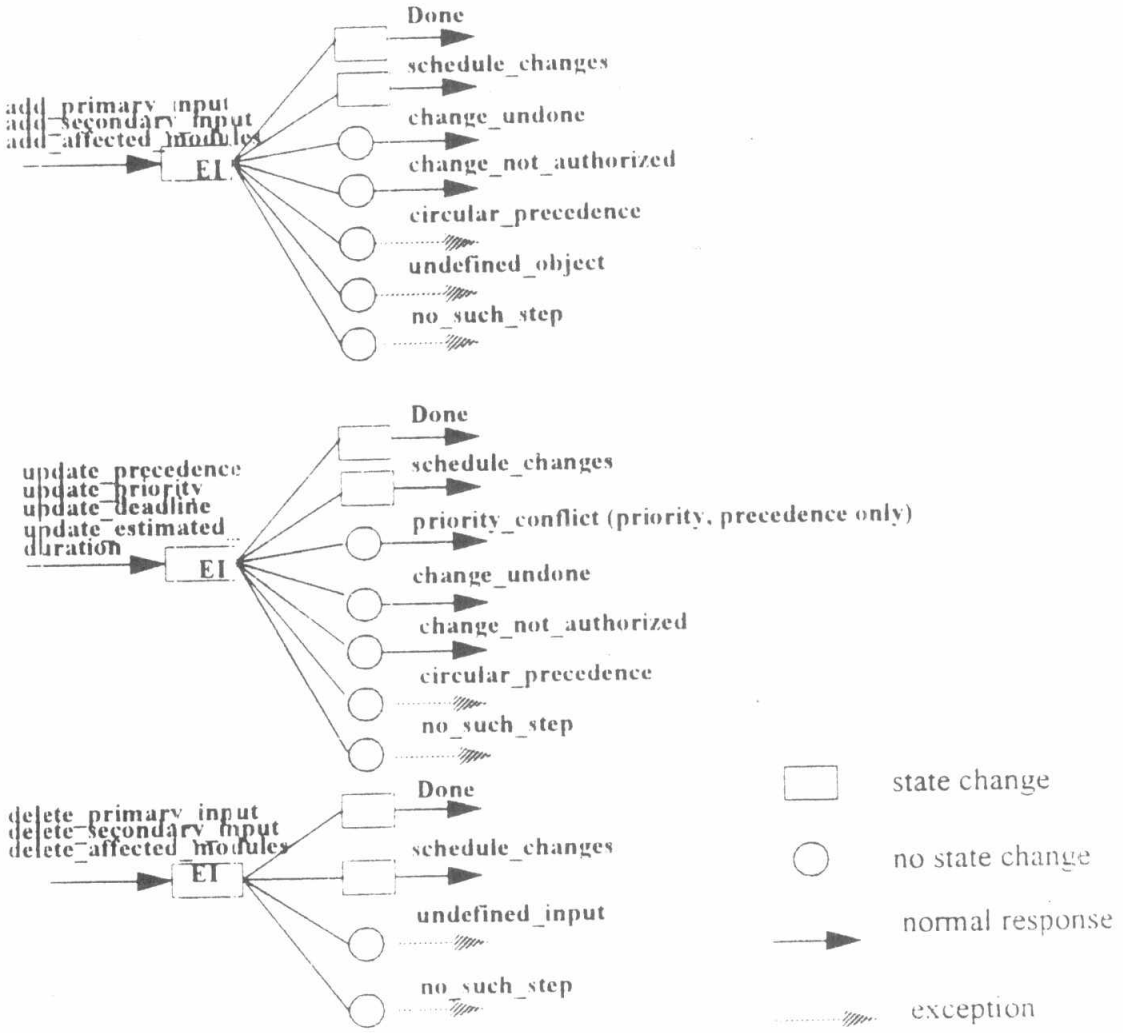

FIGURE 5. Stimulus Response diagram for the edit interface

E. CONCLUSION

Integrating planning and version control allows both parts to be more fully automated. This technology may also enable a single manager to handle projects of larger size by providing decision support and taking care of low level details.

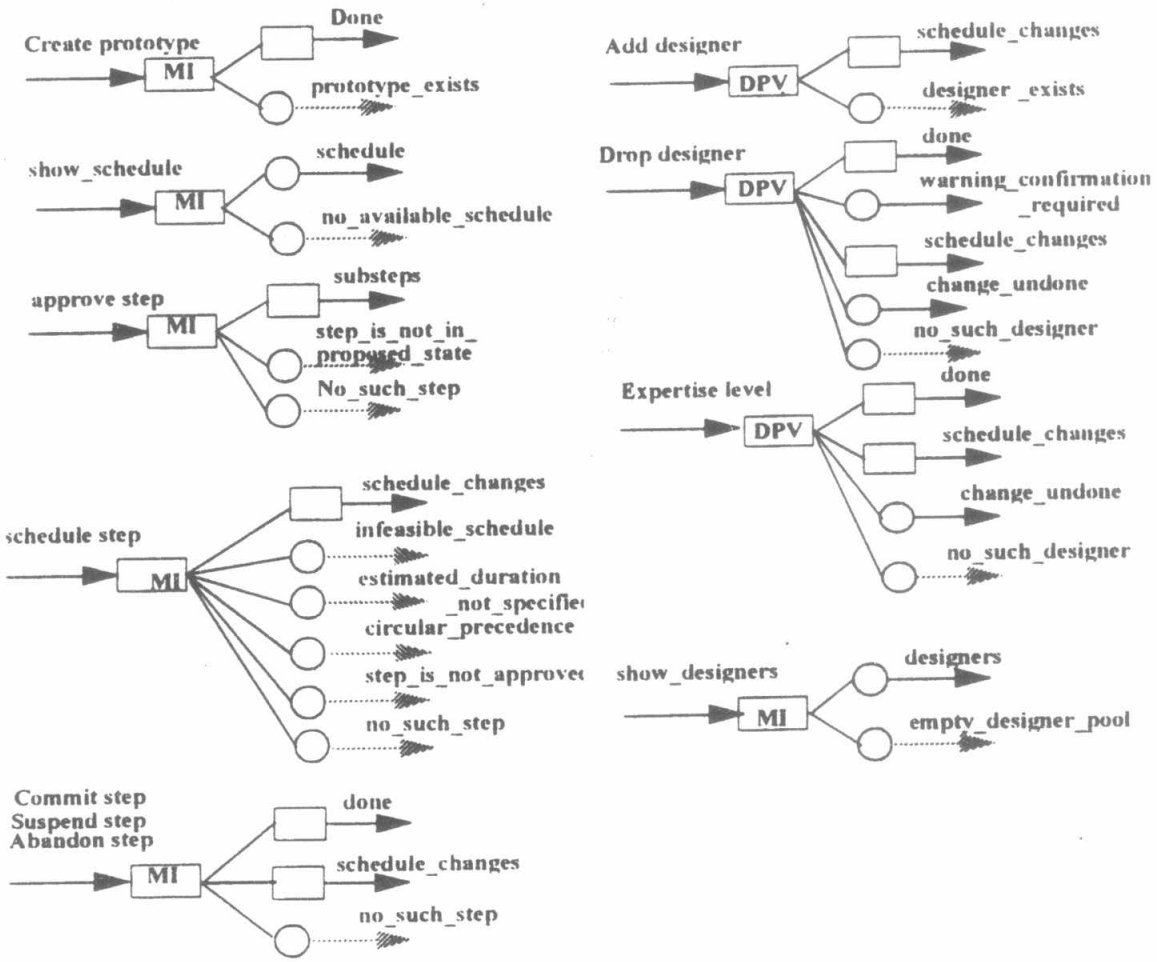

FIGURE 6. Stimulus Response diagram for the manager interface

REFERENCES

[1] Badr Salah, "A Model and Algorithms for a Software Evolution Control System", Ph. D Dissertation, Computer Science Department, Naval Postgraduate School, December 1993.

[2] Dampier D., Luqi, "A Model for Merging Software Prototypes", Technical Report, NPS CS-92-014.

[3] Feiler P. H., "Configuration Management Models in Commercial Environments", Technical Report CMU-91-TR-7, ESD-91-TR-7, 1991.

[4] Feldman S. I., "Software Configuration Management: Past Uses and Future Challenges" Proceedings of 3rd European Software Engineering Conference, ESEC '91, Milan, Italy, October 1991

[5] Hong K. and Leung J., "On-Line Scheduling of Real-Time Tasks" Real-Time Systems Workshop, May 1988.

[6] Kaiser G. E., and Perry D. E., "Workspaces and Experimental Databases: Automated Support for Software Maintenance and Evolution", Proceedings of IEEE Conference on Software Maintenance 1987. pp. 108-114.

[7] Kaiser G. E., and Perry D. E., and Schell W. M., "Infuse: Fusing Integration Test Management with Change Management", Proceedings of the Thirteenth Annual International Computer Software & Applications Conference, Orlando, FL, September 20-22, 1989.

[8] Lie A. et al, "Change Oriented Versioning in a Software Engineering Database", Proceedings of 2nd International Workshop on Software Configuraton Management, Princeton, New Jersey, Oct. 24,1989. pp. 56-65.

[9] Luqi, "A Graph Model for Software Evolution", IEEE Transaction on Software Engineering. Vol. 16 NO. 8. Aug. 1990. pp. 917-927.

[10] Mostov I., Luqi, and Hefner K., "A Graph Model for Software Maintenance", Tech. Rep. NPS52-90-014, Computer Science Department, Naval Postgraduae School, Aug. 1989.

[11] Perry D. E., and Kaiser G. E., "Infuse: A Tool for Autonatically Managing and Coordinating Source Changes in Large Systems", Proceedings of the 1987 ACM Fifteenth Annual Computer Science Conference. St Louis, Missouri, February 1987. pp 292-209.

[12] Ramamritham K., Stankovic J. A., Shiah P., "Efficient Scheduling Algorithm for Real-Time Multiprocessor Systems", COINS Technical Report 89-27, Dept. of Computer and Information Science, University of Massachusetts, 1989.

[13] Simmonds Ian, "Configuration Management in the PACT Software Engineering Environment", Proceedings of 2nd International Workshop on Software Configuration Management, Princeton, New Jersey, Oct. 24,1989. pp. 118-121.

[14] Silberschatz A., Stonebraker M., and Ullman J., "Database Systems: Achievements and Opportunities", Communication of the ACM, October 1991/Vol. 34, No. 10, pp. 110-120.

[15] Sommerville Ian "Software Engineering", Fourth edition, Addison-Wesley 1992

[16] Stankovic J. A., Ramamritham K., Shiah P., and Zhao W., "Real-Time Scheduling Algorithms for Multiprocessors", COINS Technical Report 89-47.

[17] Tichy W. F., "RCS- A System for Version Control", Software Practice and Experience, VOL. 15 (7), July 1985. pp 637- 654.

[18] Tichy W. F., "Tools for Software Configuration Management", International Workshop on Software Version and Configuration Management", Grassau, FRG 27-29 January 1988.

[19] William B. Franks, C. J. Fox, and B. A. Nejmeh, "Software Engineering in the UNIX/C Environment", Prentice Hall 1991

[20] Xu Jia., "On Satisfying Timing Constraints in Hard-Real-Time Systems", IEEE Transactions on Software Engineering, Vol. 19, No. 1, January 1993.

[21] Xu Jia., "Multiprocessor Scheduling of Processes with Release Times, Deadlines, Precedence, and Exclusion Relations", IEEE Transactions on Software Engineering, Vol. 19, No. 2, February 1993.